FUSE – Flexible File Format and Intermediate Representation for Secure Multi-Party Computation

Full Version*

Lennart Braun
Aarhus University
Aarhus, Denmark
braun@cs.au.dk

Moritz Huppert
Technical University of Darmstadt
Darmstadt, Germany
moritz.huppert@stud.tu-darmstadt.de

Nora Khayata
Technical University of Darmstadt
Darmstadt, Germany
khayata@encrypto.cs.tu-darmstadt.de

Thomas Schneider
Technical University of Darmstadt
Darmstadt, Germany
schneider@encrypto.cs.tu-darmstadt.de

Oleksandr Tkachenko
DFINITY
Zurich, Switzerland
oleksandr.tkachenko1@gmail.com

ABSTRACT

Secure Multi-Party Computation (MPC) is continuously becoming more and more practical. Many optimizations have been introduced, making MPC protocols more suitable for solving real-world problems. However, the MPC protocols and optimizations are usually implemented as a standalone proof of concept or in an MPC framework and are tightly coupled with special-purpose circuit formats, such as Bristol Format. This makes it very hard and time-consuming to re-use algorithmic advances and implemented applications in a different context. Developing generic algorithmic optimizations is exceptionally hard because the available MPC tools and formats are not generic and do not provide the necessary infrastructure.

In this paper, we present FUSE: A Framework for Unifying and Optimizing Secure Multi-Party Computation Implementations with Efficient Circuit Storage. FUSE provides a flexible intermediate representation (FUSE IR) that can be used across different platforms and in different programming languages, including C/C++, Java, Rust, and Python. We aim at making MPC tools more interoperable, removing the tight coupling between high-level compilers for MPC and specific MPC protocol engines, thus driving knowledge transfer. Our framework is inspired by the widely known LLVM compiler framework. FUSE is portable, extensible, and it provides implementation-agnostic optimizations.

As frontends, we implement HyCC (CCS'18), the Bristol circuit format, and MOTION (TOPS'22), meaning that these can be automatically converted to FUSE IR. We implement several generic optimization passes, such as automatic subgraph replacement and vectorization, to showcase the utility and efficiency of our framework. Finally, we implement as backends MOTION and MP-SPDZ (CCS'20), so that FUSE IR can be run by these frameworks in an MPC protocol, as well as other useful backends for JSON output and the DOT language for graph visualization. With FUSE, it is possible to use any implemented frontend with any implemented backend and vice-versa. FUSE IR is not only efficient to work on and much more generic than any other format so far – supporting, e.g., function calls, hybrid MPC protocols as well as user-defined building blocks, and annotations – while maintaining backwards-compatibility, but also compact, with smaller storage size than even minimalistic formats such as Bristol already for a few hundred operations.

1 INTRODUCTION

In Secure Multi-Party Computation (MPC), two or more distrusting parties jointly compute a public function on their private inputs, such that nothing beyond the output data is revealed. In recent years, researchers suggested a range of applications for MPC, such as privacy-preserving machine learning [MZ17], avoiding satellite collisions [HLOWI16], or secure auctions [BDJ+06]. In addition to the theoretical foundations starting in the 1980s [BOGW88, CCD88], several MPC frameworks have been developed. Since the development of the first MPC framework, Fairplay [MNPS04], many MPC frameworks have been introduced for running MPC protocols (e.g., [BDST22, BCS21, DSZ15, Kel20, ACC+21, HEKM11, KMSB13, DSZ15, Mal11, HKoS+10], but also for generating efficient circuits for MPC from high-level languages (e.g., [MNPS04, BDNP08, Ebr15, LWN+15, DDK+15, BDK+18, PSSY21]). An overview on different state-of-the-art MPC frameworks is provided by [HHNZ19].

Currently, the MPC research still intensively works on optimizing MPC techniques and protocols, especially for reducing the communication (e.g., [CRR21]). But the practical part of MPC becomes more and more important, since MPC gets to the point where it is deployed for real-world applications [ABL+18]. However, most MPC implementations are stand-alone and use their own internal representation for the functionality that is to be securely computed. A number of high-level languages are available that make expressing these functionalities accessible for developers with little to no knowledge about MPC protocols. These include domain specific languages, e.g., Secure Function Definition Language (SFDL) [MNPS04, BDNP08] and MAMBA [ACC+21], and even subsets of common programming languages like C [FHK+14, BDK+18] and Python [HKoS+10, Kel20]. As there are no MPC protocols available for all high-level constructs that are available in common programming languages, these functionalities have to be compiled down to less expressive internal circuit formats that effectively describe data-flows. Examples of the most popular circuit formats are

*This is the full version of our paper [BHK+23]. Please cite the conference version which will appear at AsiaCCS 2023.

Table 1: Overview of existing functionality description formats for MPC.

Format	Tools that support format	Tools for running custom optimizations	Boolean	Unrestricted #Parties	Human-readable	Function Calls	Arithmetic	Vectorization	Backwards-compatibility	Custom Annotations	Loops
Bristol Format [ST19]	[DSZ15, BDST22, Kel20, ACC+21, BDK+18, BCM+19, WMK16, HST+21]	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗
Bristol Fashion [AAL+19]	[DSZ15, BDST22, Kel20, ACC+21, BDK+18, BCM+19, WMK16]	✗	✓	✓	✓	✗	✗	(✓) MAND	✗	✗	✗
SHDL [MNPS04, BDNP08]	[MNPS04, BDNP08]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
BMR circuit format [RJHK19]	[RJHK19]	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗
SCD format [BHKR13, Ebr15]	[BHKR13, Ebr15]	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗
PCF [KMSB13]	[KMSB13]	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓
Frigate circuit format [MGC+16]	[MGC+16]	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗
MP-SPDZ bytecode [Kel20]	[Kel20]	✗	✓	✓	✗	✗	✓	✓	✗	✗	✗
HyCC circuit format [BDK+18, FHK+14]	[BDK+18, FHK+14, BDST22, DSZ15]	[BDK+18]	✓	✓	✗	✓	✓	✗	✗	✗	✗
FUSE IR (this work)	[BDST22, Kel20]	FUSE (this work)	✓	✓	✓(§A)	✓	✓	✓	✓	✓	planned

shown in Table 1. These low-level formats are much simpler than the high-level languages and were mostly developed alongside the corresponding high-level compilers [MNPS04, BDNP08, BDK+18, ACC+21, Ebr15, Kel20, MGC+16]. These circuit formats are used in state-of-the-art MPC compilers that execute the described functionality with MPC protocols [DSZ15, ACC+21, Kel20, BCS21, BDST22, Ebr15].

Currently, most of these circuit formats only support Boolean circuits. But the advances of protocol mixing [IMZ19, BDK+18, FBL+22] suggest to make use of both Boolean and arithmetic protocols for efficient MPC. Languages like [AAL+19, ST19, MNPS04, BK13] do not support modular descriptions via function calls. This makes the description of realistic functionalities memory-consuming. For example, a single instantiation of the SHA-256 compression function in the Bristol Fashion format [AAL+19] takes 3.5 MB of disk-space, whereas with preserving the structure in FUSE IR, our implementation only uses 1.1 MB of disk-space. Since the formats have been developed alongside the MPC frameworks, they suffice for the purposes of the target framework, but are often poorly documented and not extensible. Our proposed format, FUSE IR, supports both Boolean and arithmetic operations, as well as describing custom building blocks and annotations, which makes it flexible and extensible by design. It is not restricted to any number of parties, in fact, the parties do not need to be described as FUSE IR only captures the computation. It supports function calls to keep the representation compact and can be viewed through DOT graph backends.

Having so many stand-alone frameworks and formats makes problems in developing algorithmic optimizations and applications in MPC: There are many tools for both high-level compilation and compilation for MPC that are tightly coupled with each other because they share the same representation, but none of the other tools do. This leads to recurrent efforts made in each high-level compiler, where researchers have to reimplement MPC-independent optimizations like dead code elimination or constant folding. Besides that, this tight coupling also renders it impossible to reuse implementations across different frameworks or compare their efficiency without lots of developmental efforts.

Our Contributions. Our main contribution is the design and implementation of FUSE IR, a flexible intermediate representation to express functionalities for MPC that can be easily extended and is independent of any specific MPC protocol implementation. By utilizing a state-of-the-art serialization framework, FUSE IR is efficiently storable and remains compact when loaded to memory. It is cross-platform and usable in different common programming languages like C/C++, Python, Java, and Rust. With that, it is possible to generate and optimize FUSE IR inside one language, store it, and then use it from then on for applications that are developed in other languages. We have implemented a user-friendly library that enables developers to implement their own passes, i.e., analyses and optimizations. By using this library ourselves, we have implemented numerous example passes which also serve as examples for implementing new ones. To optimize generated FUSE IR, we have implemented the common data-flow optimizations Constant Folding and Dead Code Elimination. Our Frequent Subcircuit Replacement optimization pass tackles the problem that data-flow optimizations may get very large due to loop unrolling and function inlining performed by high-level compilers [BDK+18]. Our Instruction Vectorization pass transfers multiple identical Single Instruction Single Data (SISD) gates to a Single Instruction Multiple Data (SIMD) gate, which reduces the memory foot-print and the computation time in MPC [SZ13].

We also provide implementations of frontends and backends of a subset of popular MPC frameworks and circuit formats. Currently, FUSE IR can be generated from Boolean circuits in Bristol Format [ST19], mixed (arithmetic/Boolean) circuits compiled with HyCC [BDK+18], and mixed circuits in MOTION [BDST22]. FUSE IR can be executed with the state-of-the-art MPC frameworks MOTION [BCS21, BDST22] and MP-SPDZ [Kel20]. FUSE contains a Graphviz DOT backend to generate a human-readable version of our binary format. We also conduct performance analyses of memory, storage, and runtime efficiency of the FUSE framework, as well as runtime efficiency of MOTION [BDST22] when evaluating our optimized circuits. We achieve 15× faster runtimes when evaluating the Keccak_f permutation with the BMR protocol in MOTION [BDST22] and 6× less in-memory usage for our format compared to HyCC [BDK+18]. The source code of FUSE with all our benchmarks is publicly available.[1].

Outline. The rest of this paper is structured as follows. We introduce preliminary information and terminology in §2 and present the design of FUSE IR along with the architecture of our framework in §3. In §4, we present our experimental results and discuss related work in §5. We discuss future work in §6.

2 PRELIMINARIES

In this section, we present preliminary terminology and information.

[1] https://encrypto.de/code/FUSE

2.1 Secure Multi-Party Computation

In Secure Multi-Party Computation (MPC), two or more parties want to jointly compute a functionality without revealing their private inputs to the other parties. Trivially, this can be done by sending the inputs to a trusted third party and letting the trusted third party perform the computation. However, MPC enables the set of parties to carry out the computation of the functionality amongst themselves, even if such a trusted party does not exist. This process of computation is called a protocol and means that the parties compute parts locally and send messages to each other. After following the protocol, because of the properties of MPC, the parties retrieve the output but will learn nothing about the inputs of the others except what was already inferable from the output itself. There exist MPC protocols for different numbers of parties and different security models, e.g., a passive adversary is assumed to strictly follow the protocol, and an active adversary may deviate arbitrarily from the protocol specification. Many MPC protocols use Boolean, arithmetic, or mixed circuits to describe the functionality.

2.2 Circuit Compilation and Data Serialization

Generating a description suitable for MPC protocols like Boolean or arithmetic circuits by hand does not scale well. Hence, special compilers for MPC have been developed for a functionality description written in a domain-specific language [MNPS04, BDNP08, RHH14, ARG+21, LWN+15, BSM+21, MGC+16, KMSB13, SKM11] or general-purpose language like C or Python [FHK+14, Kel20, ZE15]. From that, a target representation is generated and securely evaluated by the target MPC framework during runtime. However, this compilation process is not trivial, so repeatedly running the same compilation is undesirable. To run the compilation only once, the compiled representation needs to be stored on disk and loaded later whenever that functionality is securely evaluated with MPC. These storing and loading processes are called data serialization and deserialization.

Data serialization describes the process of storing structured data that has been computed in a persistent way, such that it is restorable at a later time, preserving structure and content. Restoring the structured data with the same structure and content is called deserialization. Usually this means that structured data is stored into a stream of bytes during serialization, and deserialization restores that data with the same structure and content. We also refer to serialization as packing, and to deserialization as unpacking, because this is the common terminology used by the FlatBuffers [vO14] serialization framework, which we use in this work.

2.3 FlatBuffers

Coming up with an own method of serializing data efficiently is very tedious and error-prone, e.g., developers need to be careful with very low-level details like byte-endianness. Additionally, a custom solution is most likely only implemented inside one programming language. To support the custom serialization method in multiple projects, developers would have to rewrite parsers and generators for the programming language of their choice, which is undesirable. For these reasons, FUSE uses FlatBuffers, an open-source serialization framework maintained by Google [vO14]. FlatBuffers supports many popular programming languages, such as C/C++, Go, Java, Rust, and Python. The most popular examples of projects that use FlatBuffers is the popular Cocos2d-x[2] mobile game engine and Facebook's Android client[3], where Facebook improved the performance of their client drastically by changing JSON serialization to FlatBuffers. The main performance improvement lies in the so-called zero-copy overhead: FlatBuffers allows to read serialized data without parsing or unpacking it into custom data structures, making the process of reading serialized data extremely efficient.

Serialized data is called a *FlatBuffer* which essentially is a binary buffer that consists of nested objects like vectors and structs. These objects are organized with offsets, which enables random access to the elements inside a FlatBuffer. The structure, from which the offsets are derived, is described in a so-called schema file. The schemas describe the types used inside the buffer. Since all the values inside the buffer are strictly typed, the binary data is traversable in-place, which in turn makes read access very efficient. FlatBuffers defines strict alignment and endianness rules for in-memory data, making the serialized data cross-platform.

Given the schema, the `flatc` compiler generates special code that handles the read access to the binary buffer, which we will refer to as accessor code. The compiler generates this code for all the supported programming languages. With this, developers only need to define the schema once, and the compiler will generate all the necessary implementations for the given serialization schema. It also supports a restricted form of format evolution. This means that the structure of the binary defined by the schema may be extended, and the resulting code will still be able to handle binaries that have been produced before the schema evolution.

3 THE FUSE FRAMEWORK

This chapter presents the design and implementation of our FUSE framework. We first discuss the motivation behind the design of our intermediate representation FUSE IR together with the syntactic aspects of the language and explain the intended semantics behind them (§3.1). Building on top of this language, we present the system architecture of the whole FUSE framework (§3.2) and present frontends (§3.4), backends (§3.5), and optimization passes (§3.6).

3.1 FUSE Intermediate Representation (IR)

FUSE IR decouples MPC-specific details from the actual functionality that is computed. For example, to describe a functionality with Boolean operations, we can omit naming a specific protocol for every operation. If the exact protocol needs to be stored, our format supports custom annotations at every description level.

The main considerations behind FUSE IR are:

- FUSE IR only allows describing data oblivious programs. Because of that, it allows describing Boolean and arithmetic operations and loops with a public number of iterations, but no control-flow elements like if-else statements.
- It is a modular representation with support for function definitions and calls, as well as loops with public number of iterations.
- FUSE IR is extensible, meaning that its specification can be easily adapted to support new operations. When extending

[2] https://github.com/cocos2d/cocos2d-x

[3] https://engineering.fb.com/2015/07/31/android/improving-facebook-s-performance-on-android-with-flatbuffers/

the specification with new operations, FUSE will still support previously compiled FUSE IR with no adaptions needed.
- To make the representation more flexible, we allow describing custom operations and optional custom annotations at every level. This enables MPC developers to adapt FUSE IR to their needs and still use it within the whole FUSE framework.

Figure 1 shows the definition of FUSE IR as FlatBuffers [vO14, vO17] schemas. This is both a description of what elements can be used to define a program in FUSE IR and how the representation is stored when serialized. We present an example FUSE IR module in §E

Modules. In FUSE IR, a module is the top-level entity of description. It contains a list of circuits and may declare one of these as the entry point for computation. The entry point marks the circuit which shall be executed when the whole module is executed. The list of circuits are then used to resolve function calls during execution. This is similar to the Executable and Linking Format (ELF), which also allows to dynamically resolve procedure calls. A circuit in turn has a name which identifies it inside the module, and holds a list of nodes which describe the computation.

Nodes. Each node has a unique identifier, so the circuit uses the identifiers of the nodes to mark global input and output nodes of the circuit, together with the types of inputs of outputs. A node also defines its input nodes, the operation it performs on the inputs, and the number of outputs with their data types. For the inputs, a node lists all identifiers of its inputs and can also define offsets to select a specific output of a node. This is important whenever nodes produce more than one output. The operation that is performed on the inputs is stored in the `operation` field, where all currently supported operations are listed in the `PrimitiveOperation` enumeration, which can be expanded in the future. Alternatively, the `Custom` operation type can be used in connection with an annotation.

Schema Details. The main elements of descriptions are the schemas for `ModuleTable`, `CircuitTable`, `NodeTable`, and `DataTypeTable`. These are so-called FlatBuffers tables which describe objects in FlatBuffers, and consist of a list of fields with the corresponding data types. For example, the `ModuleTable` schema declares the three fields `circuits` which holds a list of `CircuitTableBuffer`, `entry_point` which is of type `string`, and `module_annotations` which is also of type `string`. We emphasize that an instance of this table does not need to define all of these fields, in which case they will not be stored in the binary. Additionally, as schemas can always be safely extended with more fields, we can add more descriptive fields to the schema and the whole serialization and deserialization process will remain intact. Of course, in some cases the code will still need adaptions for new description elements; however, it does not entail low-level implementations for serialization, so researchers can instead focus on the functionality of their frameworks. Between the description of a module and a circuit, there is a special `CircuitTableBuffer` table that stores already serialized circuits. This implements an indirection mechanism that enables us to define and store circuits that are not part of a module. Otherwise, a circuit would always have to be stored inside a module, even if it would be the only one inside there. Additionally, this indirection allows us to deserialize only selected circuits. For our use case, this allows us to keep the memory footprint as small as possible when we optimize only a subset of circuits inside a module, so the unmutated circuits can remain serialized and we only unpack the circuits to be optimized. This table is shown here for the sake of completeness, but this indirection is taken care of internally, without direct exposure to developers.

Annotations. As different tools may have the need for more description in the functionality which can be unsuited to add to the description of FUSE IR, each table defines an optional annotation field, where developers can store such additional descriptions. However, these are not necessary inside the FUSE infrastructure and are ignored during optimizations.

3.2 FUSE Architecture

FUSE IR is the heart of our FUSE framework which is built around it. The components and structure of FUSE are visually depicted in Figure 2. The box "FlatBuffers schemas of FUSE IR" in the top left of the figure refers to the schema definitions from Figure 1. These are compiled via the `flatc` compiler to C++ accessor code for reading and generating FUSE IR in C++. This accessor code handles reading and writing the binary buffers.

FUSE Core (§3.3). Because the accessor code is automatically generated, we introduce an abstraction layer, called the FUSE Core, in between the definition of FUSE IR and the actual frontends, optimization passes and backends for FUSE IR. It defines a user-friendly application programming interface (API) to read, mutate, and write FUSE IR compactly without worrying about the serialization details of FlatBuffers. Additionally, if in the future the schema definitions of FUSE IR change, the accessor code will change as well, so the FUSE Core would be the only part that needs to adapt while keeping the API for the rest of the framework as similar as possible.

FUSE Frontends (§3.4). We implement frontends to generate FUSE IR from other circuit descriptions. Currently, there is a frontend for the textual Bristol Format [ST19] (§3.4.1), the hybrid compiler HyCC [BDK+18] (§3.4.2), and mixed-circuits from MOTION [BDST22] (§3.4.3)).

FUSE Backends (§3.5). The FUSE Backends export our intermediary language FUSE IR to other representations, or evaluate it, either in plaintext or with the state-of-the-art MPC frameworks MOTION [BDST22, BCS21] (§3.5.1) and MP-SPDZ [Kel20] (§3.5.2).

FUSE Optimization Passes (§3.6). As tutorial examples, we have implemented several optimizations available for FUSE IR, which are currently the general-purpose data-flow optimizations Constant Folding and Dead Code Elimination (§3.6.1), as well as the more MPC-specific optimizations Instruction Vectorization (§3.6.2) and Frequent Subcircuit Replacement (§3.6.3).

3.3 FUSE Core

The FUSE Core serves two main purposes: Firstly, it removes the coupling between frontends, passes, and backends of FUSE and the generated accessor code from FlatBuffers [vO14]. If the schemas for FUSE IR change, the generated code will change as well and might result in breaking the complete code base. Now, with schema changes only the FUSE Core has to be adapted, which can be done in a way that does affect nothing or fewer parts of the entire code base. Secondly, the C++ API for FlatBuffers contains many details about the serialization process that need to be kept in mind when

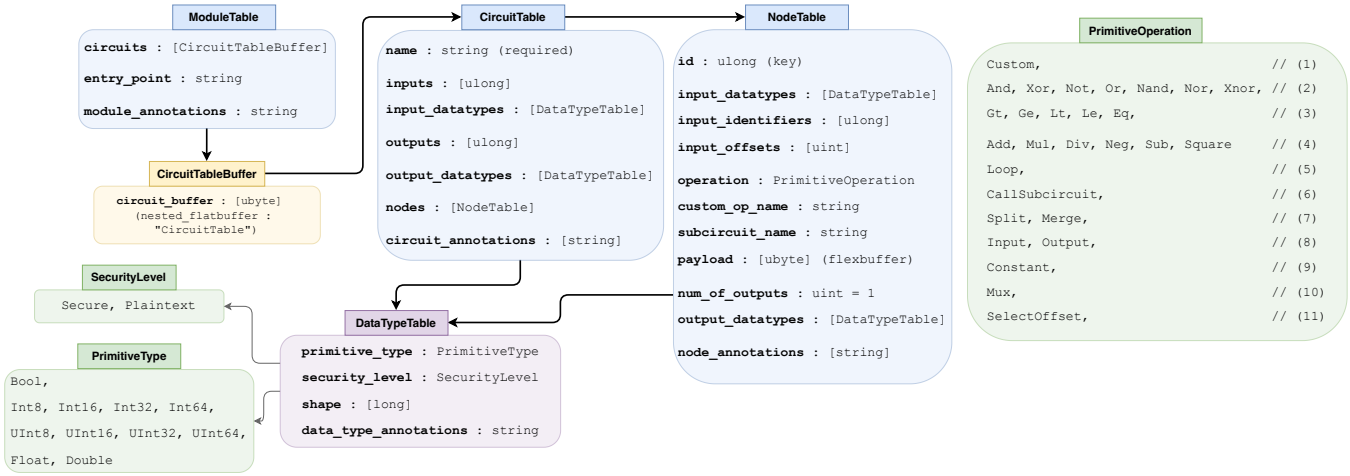

Figure 1: FUSE IR as FlatBuffers [vO14, vO17] schema description.

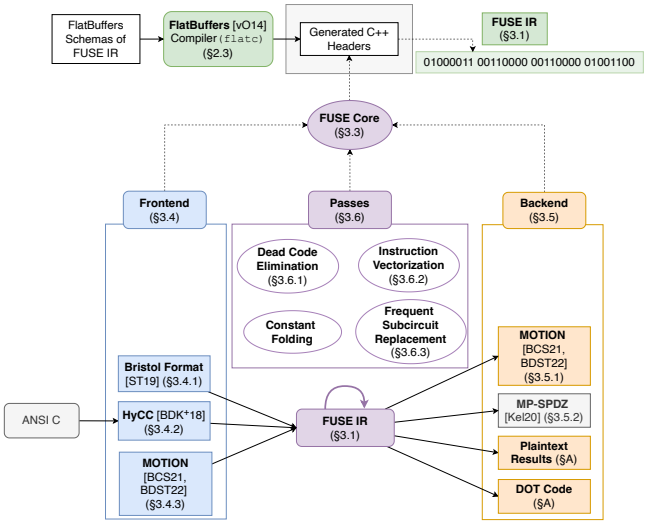

Figure 2: System architecture of the FUSE framework.

using as a developer. For example, when serializing an object, we have to serialize all the object's field and then call the serialization routine for the whole object. For larger data structures, especially in our case, this makes the whole process too confusing and thus, impossible to use. To make the FUSE library more flexible and easy to use, we introduce an additional level of abstraction that is agnostic to which representation is used underneath, e.g., if internal FlatBuffers types are used or C++ native types.

Moreover, the Builder API in FUSE Core can also be used to programmatically create a program in FUSE IR directly instead of converting it from a different representation via a Frontend.

Since the FUSE Core layer has been implemented in C++, its abstractions from FlatBuffers are not available in other programming languages. However, it makes sense for developers to write optimizations in C++ using the abstraction layer and only parts that have to be implemented in the other programming language with the generated accessor code for that language to read the optimized FUSE IR. In that case, it is important to keep in mind that the code parts with dependencies on the generated FlatBuffers API may be subject to change.

In §2.3, we have introduced the notion of serialized and unpacked data. Serialized data refers to binary data containing the FlatBuffers data, whereas unpacked data refers to the same data parsed into special data structures, in our case C++ objects. Unpacking is needed if we want to do complex mutations on our binary FUSE IR data. Simple cases, e.g., changing the value of an integer, can be realized in the binary directly, if the field was present in the binary before. However, this case might not work if the field was missing before because it might not have been explicitly stored in the binary. Especially when we want to change more than a single integer field, e.g., performing optimizations on FUSE IR, working on the binary data is insufficient. For these cases, we unpack the serialized data into C++ objects with data structures that are also automatically generated by FlatBuffers. To keep the treatment of the data as similar as possible, the FUSE Core also defines an interface to both serialized and unpacked data. This is useful for analysis passes and backends, as these only require read access to FUSE IR, so they can technically work with both serialized and unpacked data. Without that interface, developers would need to rewrite the analysis or backend for both types of data, which we clearly omit.

To summarize, the FUSE Core contains: (1) A Builder for generating FUSE IR. (2) Wrappers that unify read access to serialized and unpacked data and simplify mutating FUSE IR. (3) A Context that is responsible for handling serialized and unpacked data, as well as coordinating serialization and deserialization. (4) C++ type traits and policies for many of the supported operations to simplify template programming with FUSE IR.

3.4 FUSE Frontends

A frontend reads a functionality description in the source format and generates an equivalent version in FUSE IR. By implementing frontends for other formats, they can be used with the rich infrastructure of the FUSE framework: The functionality can be executed

with the available MPC backends (§3.5), and optimizations (§3.6) can be applied that might be unavailable for the source format.

3.4.1 Bristol Frontend. Bristol Format [ST19] is a human-readable, text-based file format which describes a stand-alone Boolean circuit inside a single file. It describes two-party computation, but was extended to an arbitrary number of parties in the very similar Bristol Fashion [AAL+19]. A file in Bristol Format consists of a header that defines the circuit interface, and the actual functionality description that lists all the gates and their wiring in the circuit. The format supports the Boolean AND, XOR, and INV operation, which we can directly translate to FUSE IR. While this translation can increase the size of the representation, it enables us to perform analyses, optimizations, and use the complete FUSE toolchain for the circuit description originally provided in Bristol Format.

3.4.2 HyCC Frontend. HyCC [BDK+18] is a high-level compiler that compiles ANSI C code to hybrid circuits with both Boolean and arithmetic gates and support for function calls. The circuit description is stored in the custom hand-crafted binary `.circ` file format, which is optimized for space-efficiency. HyCC provides tools to read them directly into their own in-memory representation in C++, but the format itself is only documented inside the source code[4]: A format header describes the circuit interface, with declarations for input/output variables and function call interfaces. It is followed by the wiring of the circuit with gate definitions, output variables, and inputs for function calls. We translate the HyCC format into a single FUSE IR module, that contains all translated circuits. For function calls, we first translate the called subcircuit (if not already done) and then reference it with a call node in the FUSE IR.

3.4.3 MOTION Frontend. MOTION [BDST22, BCS21] is a generic C++ framework for efficient mixed-protocol MPC and designed to be extensible with more MPC protocols in the future. Currently, several multi-party protocols with passive security are available inside the framework, including arithmetic and Boolean GMW [GMW87], constant-round BMR [BMR90], and protocol conversions. In MOTION, the functionality is described by gates that also encode the underlying protocol to execute that gate. The resulting secret-shared values of the computation are stored in wire objects.

Gates and wires are implemented as C++ classes for each combination of protocol and supported gate type. So, every gate describes an operation and provides output wires and most of the gates additionally contain inputs. An exception are constant gates that only produce a constant output wire. Hereby, each gate translates to a computational node in FUSE IR with the same operation as the gate. To correctly connect the nodes in FUSE IR, each wire in MOTION is mapped to the output of a node in FUSE IR. A node output can be described by a pair `(Identifier, Offset)`. The Identifier refers uniquely to the specific node and the Offset refers to one of the node's outputs. For example, the second output of the node with Identifier 4 is represented by `(4,1)`.

One additional feature that MOTION has compared to the previous formats are Single Instruction Multiple Data (SIMD) values. These are used to compute a single gate with multiple values in parallel, which improves communication and computation time significantly in MOTION [BDST22]. SIMD values are realized inside wires, which means that a single wire stores one or multiple values. The values can either be initialized inside input gates or using specific `Simdify` gates which construct SIMD values from single values. To read a single values from a SIMDified value, special `Unsimdify` gates have to split the SIMD values into single values. This means that during translation, we have to keep track of the amount of SIMD values during initialization by storing the size of the SIMD value into the input node. For `Simdify` and `Unsimdify` gates, we introduce corresponding custom nodes.

Note that custom nodes cannot be used outside the context they were introduced in, unless there are translations for this specific instruction available. However, this case is justified by the fact that SIMD values are not necessarily implemented by all the frameworks that FUSE supports and might support in the future.

3.5 FUSE Backends

To generate other data or formats from a given FUSE IR description, we have implemented several backends for FUSE IR. We identify two different types of backends. (1) Interpreter backends, that evaluate the circuit described in FUSE IR either in plaintext or by using MPC protocols. (2) Code Generation backends, that generate another kind of structured data out of FUSE IR. Due to space restrictions, we describe the implemented FUSE backends related to usability in §A.

3.5.1 MOTION Interpreter. To execute FUSE IR with state-of-the-art MPC protocols, our MOTION backend evaluates the described functionality in FUSE IR with the passively secure MOTION framework [BDST22, BCS21]. It works similar to the plaintext interpreter described in §A, except that it is evaluated on MOTION shares instead of plaintext values. These shares come with a handy wrapper that enables writing this backend in an interpreter fashion without worrying about the specific MPC protocol. MOTION share wrappers implement C++ operators for the operation to be executed, such as addition of shares via the +-operator. Internally, MOTION resolves these operations by executing them with the corresponding protocol that is used for MPC at runtime.

3.5.2 MP-SPDZ Code Generation. Another way to securely execute FUSE IR is the state-of-the-art MP-SPDZ framework [Kel20] which includes actively secure MPC protocols. MP-SPDZ defines two interfaces, a high-level one in Python and a low-level one in C++. The Python interface provides custom data types like secure integers and defines operations on those similar to the MOTION share wrappers. The low-level interface handles the execution of the underlying protocol and must be used with care. This makes the C++ interface difficult to use, and its usage is not recommended by its developers. As we implement the backend with the Python interface of MP-SPDZ, we have no access to our C++ wrappers that we have carefully designed and implemented in FUSE Core (see §3.3). Instead, we compile and use FlatBuffers accessor code for Python from our FlatBuffers schemas (§1). Consequently, this is the only part of the FUSE framework that directly exposes FlatBuffers outside the FUSE Core. This is justified because the MP-SPDZ backend represents a relatively small part of the whole FUSE framework.

[4] https://gitlab.com/securityengineering/HyCC/-/blob/master/src/libcircuit/simple_circuit_file.cpp

Accessing the C++ code directly from Python would have complicated the implementation, so exposing FlatBuffers turned out to be the most user-friendly possibility.

We implement an import functionality for FUSE IR inside MP-SPDZ, similar to its circuit import functionality for Bristol Fashion [AAL+19] circuits. For this, we create a new `FuseCircuit` class that extends the existing `Circuit` class in MP-SPDZ. FUSE IR can then be imported into MP-SPDZ code and run on data that is declared inside the framework. Running a circuit on the input data starts the interpretation, whereas a module forwards the inputs to the entry circuit and then runs it. Then, the circuit is evaluated by reading and adding mappings from node identifiers to MP-SPDZ values. To utilize the secure types of MP-SPDZ, the Python code must be set up accordingly. Given this setup, FUSE IR can be used easily inside MP-SPDZ programs.

3.6 FUSE Optimization Passes

The FUSE framework provides a rich infrastructure for MPC, including an expressive intermediate representation and many circuit optimizations. Besides domain-independent optimizations such as Dead Code Elimination and Constant Folding [ALSU06], we have developed two optimizations specifically tailored to MPC: Instruction Vectorization (§3.6.2) and Frequent Subcircuit Replacement (§3.6.3).

Listing 1: Dead Code Elimination on FUSE IR Example in C++. The data structures marked in purple and the functions marked in sand color are defined in the FUSE Core (§3.3).

```
#include <IR.h>

void visit(fuse::core::NodeObjectWrapper& node,
           std::unordered_set<uint64_t>& liveNodes) {
    for (auto inputNodeID : node.getInputNodeIDs()) {
        // check if nodes were already visited and marked
        if (!liveNodes.contains(inputNodeID)){
            liveNodes.insert(inputNodeID);
            visit(inputNode, liveNodes);
        }
    }
}

void eliminateDeadCode(fuse::core::CircuitObjectWrapper& circuit) {
    // mark live nodes
    std::unordered_set<uint64_t> liveNodes;
    for (auto outputNodeID : circuit.getOutputNodeIDs()) {
        liveNodes.insert(outputNodeID);
        auto outputNode = circuit.getNodeWithID(outputNodeID);
        visit(outputNode, liveNodes);
    }
    // delete nodes that have not been marked as live
    // -> considered dead nodes
    circuit.removeNodesNotContainedIn(liveNodes);
}

int main(void) {
    fuse::core::ir::CircuitContext ctx;
    ctx.readCircuitFromFile("path/to/circ");
    fuse::core::CircuitObjectWrapper circuit = ctx.getMutableCircuitWrapper();
    auto before = circuit.getNumberOfNodes();
    eliminateDeadCode(circuit);
    auto after = circuit.getNumberOfNodes();
    std::cout << "Eliminated: " << (before - after) << " nodes from circuit "
              << circuit.getName() << std::endl;
    return 0;
}
```

3.6.1 Tutorial Pass - Dead Code Elimination. To showcase how to implement passes using FUSE Core (§3.3), we provide an example implementation for Dead Code Elimination on FUSE IR. Listing 1 shows the C++ code of executing a dead code elimination pass.

The main function starts with reading a FUSE IR circuit from a file and creating a mutable circuit object, since the optimization will mutate the circuit in-place. Then the `eliminateDeadCode` function is called which starts marking all output nodes of the circuit as live and then traverses the circuit backwards with the `visit` function. The `visit` function marks all input nodes as live – as they reach a live output node – if they have not been marked already and visits them. After the marking phase, the set containing the identifiers of all live nodes is passed over to the `removeNodesNotContainedIn` library function, which is defined as a part of the FUSE Core (§3.3). This function is responsible for deleting all the nodes whose identifiers are not inside the `liveNodes` set, which in this context means removing all dead nodes.

We stress that writing this pass takes *less than 30 lines of C++ code* and is *completely oblivious about the serialization details* of FUSE IR, because of the *abstractions* provided by FUSE Core (§3.3).

3.6.2 Instruction Vectorization. Instruction Vectorization (IV) replaces multiple identical subcircuits operating on single values with a subcircuit that operates on vectors of values. This can greatly improve the overall computation time and the memory footprint [SZ13].

The process of Instruction Vectorization assumes that multiple identical SISD-gates (Single Instruction Single Data) can be executed faster in-parallel and thus, only groups of gates that implement the same operation and lay "close" in the circuit are suitable. Our Instruction Vectorization uses two kinds of depth information to approximate the distance between gates, and to replace suitable groups of gates with a single vectorized gate. We give the full details of our Instruction Vectorization in Appendix §C.

3.6.3 Frequent Subcircuit Replacement. Many MPC circuit descriptions, such as Bristol Format [ST19, AAL+19] and the Secure Hardware Definition Language (SHDL) [MNPS04, BDNP08], do not support subroutines. These formats store reoccurring instruction sequences redundantly for every occurrence. Consequently, the description lacks memory-efficiency and hierarchical structure. In contrast, FUSE IR stores reoccurring instruction sequences in a single subcircuit and uses subroutine calls to make the intermediate representation more memory-efficient for frequently reoccurring sequences.

The FUSE framework provides frontends that translate a variety of circuit descriptions into the intermediary language (§3.4). However, if we translate a description format that does not support subroutine calls, we cannot directly make use of subroutines because the reoccurring instruction sequences lack annotation: We first need to locate the occurrences before we can create subroutines and insert subroutine calls. For this, we implement the Frequent Subcircuit Replacement (FSR) optimization that detects all occurrences of frequent instruction sequences in the circuit and replaces each sequence by a subroutine call. We transform a large non-modular circuit into a smaller representation that effectively uses the modular capabilities of FUSE IR. For this, we utilize the state-of-the-art Frequent Subgraph Mining algorithm DistGraph [TZ16]. Both stages of this algorithm are described in more detail in §D. Note that Frequent Subcircuit Replacement might also reveal unknown patterns in the MPC circuit.

4 EVALUATION

We evaluate our framework on two servers, each equipped with an Intel Core i7-4790 processor and 32 GB RAM, connected via a 10 Gbps network. We compare FUSE IR against other MPC file formats and demonstrate the usefulness of our unique optimization passes (§3.6). We have chosen Bristol Format (§4.1) as a simple textual format for Boolean circuits and the binary HyCC circuit format (§4.2) for hybrid circuits to compare FUSE IR against different types of formats. We also showcase how the MOTION [BDST22, BCS21] framework profits from our vectorization optimization (§4.3). Because FUSE can be easily extended to support more MPC backends, these results are not limited to MOTION only.

4.1 Disk Usage for Boolean Circuits

We compare the textual Bristol Format [ST19] with FUSE IR. For the details on Bristol Format, see §3.4.1. FUSE IR is more expressive than Bristol Format, because it supports more operators with no limitations on the number of inputs and outputs, as well as a modular description. While Bristol Format, being a text-based format, is human-readable by design, FUSE IR offers through its DOT backend (§A) more visual information on the structure of the circuit besides the node operations. Thus, it is easier to visually spot the structure of a circuit than with text-based information only.

Table 2: File sizes of circuits in Bristol Format [ST19] and their translations to FUSE IR.

Circuit	Bristol Size (KB)	FUSE IR Size (KB)	Zipped Bristol Size (KB)	Zipped FUSE IR Size (KB)	Improvement Factor	Zipped Improvement Factor
int_add8	0.6	3.8	0.2	1.1	0.2	0.2
int_mul8	2.4	9.1	0.7	2.3	0.3	0.3
int_add64	6.1	30.9	1.6	7.4	0.2	0.2
int_mul64	264.9	627.4	72.2	137.3	0.4	0.5
aes_128	906.9	1 918.7	239.6	421.5	0.5	0.6
aes_256	1 266.9	2 650.2	336.1	586.0	0.5	0.6
sha_256	3 557.0	7 085.0	925.2	1 576.5	0.5	0.6
sha_256_fuse[†]	3 557.0	1 107.0	925.2	191.2	3.2	4.8

[†] This row compares our implementation of SHA-256 directly in FUSE IR with the circuit file in Bristol Format.

As both formats aim to offer a compact description of circuits, we compare the file sizes of a given Bristol Format circuit and the file size of FUSE IR which was generated using the frontend for Bristol Format (§3.4.1). The results are shown in Table 2. The first two columns correspond to the file sizes of the given Bristol Format circuit and its FUSE IR translated version. The following two columns show the file sizes of both files compressed with gzip. The final two columns depict the ratio of the FUSE IR file size to the Bristol Format file size. For both compressed and uncompressed files, we observe that Bristol Format files are more compact than FUSE IR, however, the overhead of FUSE IR decreases with larger circuit sizes. Both formats use integers to describe gate/node identifiers. The Bristol Format encodes them using a decimal reprensentation in ASCII, whereas FUSE IR uses binary 64-bit integers. Hence, the Bristol Format version of describing gate identifiers uses up more space, the larger the circuits get, because the gate identifiers get longer, and Bristol includes whitespace to make it human-readable.

The last row of Table 2 shows the sizes of the SHA-256 compression function, that we have manually implemented in FUSE IR using the Builder API in FUSE Core. For our manual implementation, we have used mixed Boolean and arithmetic operations and realized shifts and rotations as subroutines. When compared to Bristol Format, that only uses Boolean operations and no subroutines, our format is over 3× more compact, despite the directly translated file being up to 2× larger. This result highlights the relevance to use FUSE IR for compact representation of larger functionalities.

Improve Storage with Frequent Subcircuit Replacement. To improve the storage efficiency for FUSE IR, we use our Frequent Subcircuit Replacement (§3.6.3) optimization. We measured the number of nodes before and after applying FSR to the circuit, as well as the file sizes of the stored FUSE IR before and after the optimization. The results are shown in Table 3. Although all of the circuits get nodes replaced, which is indicated by the last column that shows an improvement for every circuit, this does not imply that the storage usage decreases as well. There are two reasons for this: (1) For small circuits (< 1000 nodes), creating a subcircuit that is called only once or twice does not make up for the storage imposed by that subcircuit. The subcircuit created is typically larger, i.e., contains more nodes, than the pattern it replaces, because it contains additional input and output nodes. See §3.6.3 for the implementation details.

Table 3: Size Impact and replaced number of nodes with FSR.

Circuit	FUSE IR Size (KB) before FSR	FUSE IR Size (KB) after FSR	#Nodes before FSR	#Nodes after FSR	Storage Improv. Factor	#Nodes Improv. Factor
int_add8	3.9	8.0	58.0	37.0	0.5	1.6
int_mul8	9.2	10.2	160.0	115.0	0.9	1.4
int_add64	31.0	28.7	506.0	316.0	1.1	1.6
int_mul64	627.4	485.3	11 976.0	7 756.0	1.3	1.5
aes_128	1 918.8	1 544.2	37 047.0	23 481.0	1.2	1.6
aes_256	2 650.2	2 132.0	51 178.0	32 485.0	1.2	1.6
sha_256	7 085.1	5 661.1	136 097.0	81 089.0	1.3	1.7

Table 4: Size Impact and replaced number of nodes with IV.

Circuit	FUSE IR Size (KB) before IV	FUSE IR Size (KB) after IV	#Nodes before IV	#Nodes after IV	Storage Improv. Factor	#Nodes Improv. Factor
int_add8	3.8	4.2	58.0	45.0	0.9	1.3
int_mul8	9.1	7.2	160.0	58.0	1.3	2.8
int_add64	30.9	33.5	506.0	381.0	0.9	1.3
int_mul64	627.4	317.2	11 976.0	562.0	2.0	21.3
aes_128	1 918.7	910.5	37 047.0	825.0	2.1	44.9
aes_256	2 650.2	1 255.6	51 178.0	1 118.0	2.1	45.8
sha_256	7 085.0	3 682.0	136 097.0	9 417.0	1.9	14.5
Keccak_f	9 924.0	4 471.7	195 286.0	3 416.0	2.2	57.2
sha_512	18 292.3	9 251.6	351 665.0	18 721.0	2.0	18.8

Improve Storage with Instruction Vectorization. Another possible way to reduce the stored circuit size is by vectorizing as many nodes as possible in a circuit using our Instruction Vectorization optimization (§3.6.2). This reduces the metadata that needs to be stored

for each node by grouping as many single nodes together as possible. We compare the file sizes and number of nodes inside the circuits before and after applying Instruction Vectorization. The results are shown in Table 4. Since we perform a greedy optimization, meaning that we vectorize all nodes with the same operation in the same layer of that operation (see §3.6.2 for details), the number of nodes in the circuits decrease drastically, with Keccak_f having 57× less nodes than before. The storage usage of the circuits improve for circuits that contained more than 600 nodes before vectorization. Smaller circuits use up to 10% more storage because vectorized nodes add a small description overhead which is visible in these cases. The overhead stems from introducing offsets to address the correct output in the vectorized node, and these offsets introduce a slight memory overhead. However, as is visible for larger circuits, this overhead is still surpassed by the enormous reduction in the number of nodes in the circuit after the optimization.

4.2 Memory Usage for Hybrid Circuits

Table 5: Upper bounds for the peak heap usage in KB measured with massif.

Circuit	HyCC	FUSE IR	Improvement Factor
biomatch1k	≤15 289	≤6 388	2.39
biomatch4k	≤28 405	≤6 388	4.45
tutorial_euclidean_distance	≤37 702	≤12 684	2.97
gauss	≤58 939	≤12 684	2.33
kmeans	≤73 621	≤25 267	2.91
cryptonets	≤182 384	≤50 433	3.62
mnist	≤298 062	≤50 433	5.91

In this section, we compare FUSE IR with circuits generated by the HyCC [BDK+18] compiler for hybrid circuits. We translate hybrid circuits generated by HyCC to FUSE IR using our HyCC frontend (§3.4.2), and measure how compact each representation is when loaded into memory.

We use the massif heap profiler[5] to estimate memory usage of both formats. Table 5 shows the sizes of each in-memory representation. As massif measures the size of the padded heap instead of the actually allocated amount of Bytes, the numbers represent upper bounds on the actual memory that needs to be allocated for each circuit. Although the numbers do not depict the exact memory used up by the in-memory representation, measuring the peak heap usage gives a good estimate on how much RAM will be at least needed to read in the circuit and thus, use it for application. Table 5 shows the results of our measurements.

Similar to Bristol Format (see §4.1), we observed that FUSE IR circuits that were generated using the HyCC frontend (§3.4.2) result in larger files. HyCC only supports low-level operations which make the file format very compact, so we expect that manually written FUSE IR with high-level constructs will also result in more compact files. Despite that, the in-memory representation of FUSE is 2× to 6× more compact than the HyCC circuits, making it the more memory-efficient choice for use within MPC. The FUSE description could be drastically improved when higher-level operations such as vector/matrix operations are supported directly. Since we generate FUSE IR from HyCC which in turn only uses binary Boolean and arithmetic gates (see §3.4.2 for details), we do not exploit all the possibilities to make the description more compact.

[5] https://valgrind.org/docs/manual/ms-manual.html

4.3 Performance Improvements due to Instruction Vectorization

To demonstrate the possible runtime effects of our Instruction Vectorization pass (§3.6.2), we evaluate the runtimes before and after applying the optimization with the MOTION framework [BDST22]. As the circuits we use for this experiment are Boolean circuits, we evaluate the impact of our pass with the MOTION implementations of the BMR and Boolean GMW protocols.

For BMR, we compare the runtimes from the original circuits with the greedily vectorized variants from Table 4. The results are shown in Table 6. For smaller circuits (< 1000 nodes before vectorization), the runtime slightly increases after vectorization, because SIMD evaluation in MOTION incurs an extra overhead during the circuit evaluation. However, for larger candidates we can see drastic performance improvements of up to 15× for Keccak_f after applying Instruction Vectorization, as this performance overhead is overthrown by the high degree of vectorization.

Table 6: Runtimes (ms) of the BMR protocol before and after vectorizing (§3.6.2) the circuits. The runtimes are averaged over 16 runs and the maximum runtime of both parties taken.

Circuit	Circuit Evaluation Time (ms) before Vectorization	Circuit Evaluation Time (ms) after Vectorization	Improvement Factor
int_add8	151.2	150.3	1.0
int_mul8	139.3	148.8	0.9
int_add64	130.7	156.1	0.8
int_mul64	859.7	203.8	4.2
aes_128	2 131.1	254.7	8.4
aes_256	2 959.3	289.7	10.2
sha_256	7 422.8	1 572.9	4.7
Keccak_f	11 065.9	740.3	15.0
sha_512	19 149.7	3 117.4	6.1

For Boolean GMW, we observed during our experimentation that simply applying vectorization in a greedy manner does not lead to performance improvements. Instead, the vectorization needs to be performed locally. This means that the nodes inside the circuit need to have the same depth ±1 inside the circuit. So if two nodes were vectorizable in theory but have five layers of other nodes in between them, they will not be considered during vectorization. We have observed that these settings lead to better runtime performances than with greedy vectorization. But for the sake of completeness, we also include the runtimes after performing greedy vectorization.Table 7 shows the resulting runtimes for the different settings of vectorization. For the addition and multiplication circuits, the runtimes do not change much. For AES and SHA, the runtimes increase for all settings, where the higher the batch size is, the lower

the runime gets. Keccak_f is the only circuit, where vectorization improves the runtimes of the circiut by up to 10×.

5 RELATED WORK

In this chapter, we review related work to FUSE. We start with discussing compilers and frameworks for MPC (§5.1) and file formats for MPC (§5.2). Then, we summarize optimizations for MPC (§5.3). In the appendix, we discuss works from other domains: serialization frameworks (§B.2) and intermediate representations (§B.1).

5.1 MPC Compilers and Frameworks

The domain of applied MPC was pioneered by the Fairplay framework [MNPS04] which delivered a first implementation of Yao's garbled circuits (GC) protocol for two parties [Yao86]. It was extended to the multi-party BMR protocol [BMR90] in FairplayMP [BDNP08]. In addition to the implementation of MPC protocols, these tools also provided compilers for the high-level Secure Function Definition Language (SFDL) which generates the Secure Hardware Definition Language (SHDL) to describe the circuits. In a similar fashion, the frameworks Sharemind [BJL12], Frigate [MGC+16, KNR+17], Obliv-C [ZE15], ObliVM [LWN+15], and TASTY [HKoS+10] have been developed, each with different compiler frontend languages and MPC protocols for executing the program securely. The Sharemind framework [BLW08, BJL12] compiles the domain-specific language (DSL) SecreC for evaluation with their three-party hybrid protocol using additive secret sharing. Frigate [MGC+16] compiles a C-like language to a custom Boolean circuit representation, the Frigate circuit format, and evaluates it with the maliciously secure Duplo [KNR+17] GC protocol for two parties. Both Obliv-C [ZE15] and ObliVM [LWN+15] implement semi-honest two-party GC in their frameworks. Obliv-C [ZE15] extends the C language with the `obliv` keyword for secure computation, whereas ObliVM [LWN+15] compiles a Java-like DSL, ObliVM-lang, into a custom circuit representation for the ObliVM-GC backend. TASTY [HKoS+10] is the first framework that mixes multiple MPC protocols, Yao's GC [Yao86] with homomorphic encryption, and compiles from a subset of Python called TASTYL.

Ever since then, practical MPC has divided into two main areas: (1) Developing highly scalable and performant implementation of MPC protocols, e.g., [DSZ15, MR18, WMK16, BCM+19, BHKR13, Ebr15, RJHK19, Ale22, BDST22, ACC+21, Kel20, BLW08, HKoS+10, HEKM11, DGKN09, Sch18, HS13], and (2) compiling high-level function representations into optimized lower-level representations for efficient execution with MPC protocols, e.g., [FHK+14, RHH14, RSH19, ARG+21, BSM+21]. As research has shown that mixed-protocol MPC is significantly more efficient than single-protocol MPC [DSZ15, HKoS+10, BDST22, Kel20, ACC+21], a new line of research emerged with the goal to partition a program into different parts, such that each part is executed under the most efficient protocol [BDK+18, CGR+19, FBL+22, IMZ19].

In the area of MPC, the work that is most closely related to our paper is the HACCLE framework [BSM+21]. HACCLE uses the meta-programming techniques in Scala to compile the high-level DSL Harpoon to the lower-level HACCLE Intermediate Representation (HIR), which is also a DSL programmed in Scala. For evaluating HIR, the framework supports SCALE-MAMBA [ACC+21] and HoneyBadgerMPC [LYK+19] for protocols based on homomorphic encryption and Obliv-C [ZE15] for garbled circuits. Similar to FUSE IR, HIR serves as an intermediate representation between the Harpoon DSL and several backends for evaluation with MPC protocols. However, FUSE also decouples high-level compilers from the whole tool chain by introducing several frontends to FUSE IR (§3.4), which is not the case for HACCLE. The FUSE Core (§3.3) was designed to facilitate developing further frontends, backends, and passes in the future, so the extensibility of FUSE was part of its development. As we have pointed out in §3.1, FUSE IR supports extensions to the language by either providing annotations at all levels of the representations, i.e., modules, circuits, nodes, or by extending the FlatBuffers schemas that define FUSE IR (§1). Since HIR was developed as a Scala DSL, it is not serializable out-of-the-box and thus, not directly usable in frameworks written in other languages, which is the main feature of FUSE. However, FUSE does not yet provide an own frontend language, and especially does not yet implement automatic circuit conversions, where, e.g., arithmetic operations would be translated to Boolean operations. These are implemented in HACCLE and left for future work in FUSE.

5.2 File Formats for MPC

Compiling a high-level program into a highly optimized representation suitable for MPC makes the compilation process very expensive (§2.2). For this reason, several file formats have been developed to store the produced circuits persistently. Bristol Format [ST19] is the most commonly known text-based file format to store Boolean circuits for Secure Two-Party Computation (2PC). Because of its restriction to the two party setting, Bristol Fashion [AAL+19] was developed afterwards that makes small changes to support an arbitrary number of parties. Extended Bristol Fashion [AAL+19] is an additional extension to Bristol Fashion that introduces special AND gates with multiple inputs and multiple outputs, MAND gates, to describe vectorized AND gates. Bristol Format and Bristol Fashion are widely used in MPC frameworks [DSZ15, BDST22, Kel20, ACC+21, BDK+18, BCM+19, WMK16] because of their simplicity.

The Fairplay and FairplayMP frameworks [MNPS04, BDNP08] introduce the Secure Hardware Definition Language (SHDL) to describe Boolean circuits where gates are described with truth tables of arbitrary arities, meaning that gate can have two or more inputs. The BMR circuit format MPCircuits [RJHK19] supports the description of Boolean circuits with truth tables for MPC. However, it is restricted to gates that have exactly two inputs and produce one output. Similarly, the Simple Circuit Description (SCD) format [Ebr15, BHKR13] defines Boolean circuits with truth tables where each gate has two input wires. This makes describing unary functions like negation unintuitive. The Portable Circuit Format (PCF) [KMSB13] for 2PC supports Boolean circuits with function calls and a compact loop representation. Even though FUSE IR makes it possible to describe loops syntactically, the rest of the framework does not support it yet. Frigate [MGC+16] also has a Boolean circuit format for MPC that supports function calls. HyCC [BDK+18] is a compiler for MPC that compiles C to hybrid circuits consisting of both Boolean and arithmetic operations. It defines a custom binary format for 2PC, the `.circ` file format, which we have discussed in §3.4.2.

Table 7: Runtimes (ms) of the GMW protocol before and after vectorizing (§3.6.2) the circuits. The runtimes are averaged over 16 runs and the maximum runtime of both parties is taken. The batch size for n-batch vectorization refers to the minimum size of a vectorized value.

Circuit	Execution before Optimization (ms)	Execution after Greedy Vectorization (ms)	Execution after 8-batch Vectorization (ms)	Execution after 16-batch Vectorization (ms)	Execution after 32-batch Vectorization (ms)	Execution after 64-batch Vectorization (ms)	Possible Improvement Factor
int_add8	212.3	483.1	211.5	208.7	211.6	206.0	1.0
int_mul8	151.8	593.9	181.0	163.3	171.3	140.7	1.1
int_add64	139.5	2 734.8	137.2	129.9	144.4	135.6	1.1
int_mul64	588.9	5 559.1	515.3	534.7	521.0	582.2	1.1
aes_128	1 031.2	4 569.6	2 165.8	1 999.5	1 727.1	1 562.6	0.7
aes_256	1 648.1	6 214.1	2 138.4	2 389.7	2 060.8	2 163.0	0.8
sha_256	2 234.0	151 956.0	114 977.0	17 798.9	6 703.8	4 017.4	0.6
Keccak_f	5 872.8	603.7	587.5	614.4	558.8	546.5	10.8

5.3 Optimizations for MPC

The Sharemind framework [BLW08, BJL12] for Secure Three-Party Computation demonstrated substantial performance gains by introducing Instruction Vectorization (§3.6.2). By replacing multiple identical subcircuits on single bits with one n-bit subcircuit, the overall computation time and memory footprint are reduced significantly [SZ13]. With these results, state-of-the-art frameworks for MPC support Single Instruction Multiple Data (SIMD) gates for vectorization [BDST22, Kel20, BCS21, DSZ15, GIP15, KSS13, BGJK21, ZDC+21]. However, using SIMD gates requires an expert to search for suitable instructions in a large circuit, which is a tedious and laborious process. The idea for automatically vectorizing instructions inside a single circuit was suggested by [BK15] and is similar to our Instruction Vectorization algorithm (§3.6.2). GraphSC [NWI+15] introduces programming abstractions for parallelization, namely the Scatter and Gather functionalities to automatically detect parallelization.

The most recent work on vectorization was presented by Levy et al. [LSI+23] in parallel and independent work. The authors present a compiler framework that takes a Python-like routine and produces vectorized MOTION [BDST22] code. For this, the authors introduce the MPC Source intermediate language, which is a Python DSL. The vectorization uses scalar expansion and loop vectorization to produce optimized code for MOTION. This approach is orthogonal to our vectorization presented in §3.6.2, as FUSE does not perform vectorization on any high-level language, but on the generated FUSE IR directly. With FUSE, it is possible to vectorize circuits after they have already been generated, such as Bristol Format circuits [ST19]. However, since FUSE IR is a more low-level language than MPC Source, it would make sense to implement a FUSE frontend for optimized MPC Source and use the already existing FUSE backends (§3.5) and optimization passes (§3.6).

Orthogonal to the works on vectorization, expression localization reduces the amount of expressions that have to be computed with MPC [HEKM11, Ker11, Ker13]. [HEKM11] first recognized and manually applied localization, which was automatized by Kerschbaum in [Ker11, Ker13]. In [Ker11], the author presents a compiler optimization that infers program variables that can always be learnt from the input and output. Essentially, the optimization labels each program variable with the parties that know the variable's value. If a variable is known to both parties, there is no need for executing MPC protocols, so the computation can be done in plaintext which is more efficient. Building up on these results, [Ker13] presents expression rewriting rules for more efficient protocols.

In [MSY21], the authors demonstrated that optimizations are not only possible on software, but also on the hardware layer. Using Vectorized AES (VAES) instructions led to drastic performance improvements for the MPC frameworks ABY [DSZ15] and EMP-AGMPC [WMK16].

6 FUTURE WORK

Serialization Frameworks. FlatBuffers [vO14] has a size-limit for every binary buffer of around 2 GB due to implementation details. In our case, this means we cannot describe very large circuits that are larger than 2 GB. There are two ways to approach this problem: The first one is to partition large circuits such that each partition is serializable within a single FlatBuffer. This will need careful adaptions to our framework. To keep the implementation as simple as possible, the second approach is to implement FUSE IR on top of a different serialization framework. Apache Arrow [RCC+22] might be a good candidate as it allows partitioning data that is too large for memory out-of-the-box and is already used in large data-analysis projects. We discuss alternative frameworks in §B.2.
Extensions to FUSE IR. The PCF circuit format [KMSB13] proposes loop building blocks. FUSE IR theoretically supports describing loops, but there is no implementation inside the FUSE framework yet that supports them. An idea could be to encapsulate the loop body and store the number of iterations in the loop node that calls the loop body. Additionally, we could support more operations, such as operators seen in neural networks like ReLU.
Optimization Passes. In general, MPC protocols do not support all the operations that FUSE IR allows to describe. E.g., Yao's GC [Yao86] that only supports Boolean operators cannot evaluate additions out-of-the-box. For this, we could implement a specific pass that takes in a subset of operations and expresses other operators with the supported ones. Although this does not work for all possible operator subsets, it will still enhance interoperability of MPC tools using FUSE. To support more general code optimizations without re-implementing them, FUSE could use the LLVM infrastructure by implementing both a front- and backend to LLVM IR. For this, we could develop a target architecture in LLVM that

describes FUSE IR, s.t. LLVM does not generate any unsupported operations.

Acknowledgements. This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreements No. 803096 SPEC and No. 850990 PSOTI). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM).

REFERENCES

[AAL+19] David Archer, Victor Arribas Abril, Steve Lu, Pieter Maene, Nele Mertens, Danilo Sijacic, and Nigel Smart. Bristol fashion mpc circuits. https://homes.esat.kuleuven.be/~nsmart/MPC/, 2019.

[ABL+18] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases - real-world applications of secure multi-party computation. *Computer Journal*, 2018.

[ACC+21] Abdelrahaman Aly, Benjamin Coenen, Kelong Cong, Karl Koch, Marcel Keller, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE–MAMBA v1.14 : Documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation-SCALE.pdf, 2021.

[Ale22] Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. https://github.com/aicis/fresco, 2022.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[ARG+21] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *SIGPLAN*, 2021.

[BCM+19] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. Garbled neural networks are practical. Cryptology ePrint Archive, Paper 2019/338, 2019.

[BCS21] Lennart Braun, Rosario Cammarota, and Thomas Schneider. A generic hybrid 2PC framework with application to private inference of unmodified neural networks. In *NeurIPS 2021 Workshop Privacy in Machine Learning*, 2021.

[BDJ+06] Peter Bogetoft, Ivan Damgård, Thomas P. Jakobsen, Kurt Nielsen, Jakob Illeborg Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography*, 2006.

[BDK+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS*, 2018.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A system for secure multi-party computation. In *CCS*, 2008.

[BDST22] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION – a framework for mixed-protocol multi-party computation. In *TOPS*, 2022.

[BG17] Vincenzo Bonnici and Rosalba Giugno. On the variable ordering in subgraph isomorphism algorithms. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2017.

[BGJK21] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-c secure multiparty computation for highly repetitive circuits. In *EUROCRYPT*, 2021.

[BHK+23] Lennart Braun, Moritz Huppert, Nora Khayata, Thomas Schneider, and Oleksandr Tkachenko. FUSE – Flexible file format and intermediate representation for secure multi-party computation. In *AsiaCCS*, 2023.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*, 2013.

[BJL12] Dan Bogdanov, Roman Jagomägis, and Sven Laur. A universal toolkit for cryptographically secure privacy-preserving data mining. In *PAISI*, 2012.

[BK13] Niklas Buescher and David Kretzmer. Simple circuit description (SCD). https://github.com/esonghori/TinyGarble/tree/master/scd, 2013.

[BK15] Niklas Buescher and Stefan Katzenbeisser. Faster secure computation through automatic parallelization. In *USENIX Security*, 2015.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[BLZ+19] Junjie Bai, Fang Lu, Ke Zhang, et al. ONNX. https://github.com/onnx/onnx, 2019.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *STOC*, 1990.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, 1988.

[BSM+21] Yuyan Bao, Kirshanthan Sundararajah, Raghav Malik, Qianchuan Ye, Christopher Wagner, Nouraldin Jaber, Fei Wang, Mohammad Hassan Ameri, Donghang Lu, Alexander Seto, Benjamin Delaware, Roopsha Samanta, Aniket Kate, Christina Garman, Jeremiah Blocki, Pierre-David Letourneau, Benoit Meister, Jonathan Springer, Tiark Rompf, and Milind Kulkarni. HACCLE: Metaprogramming for secure multi-party computation. In *GPCE*, 2021.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *STOC*, 1988.

[CFSV18] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[CGR+19] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable and efficient secure two-party computation for machine learning. In *EuroS&P*, 2019.

[CRR21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent vole and oblivious transfer from hardness of decoding structured ldpc codes. In *CRYPTO*, 2021.

[DDK+15] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In *CCS*, 2015.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC*, 2009.

[DKS+21] Daniel Demmler, Stefan Katzenbeisser, Thomas Schneider, Tom Schuster, and Christian Weinert. Improved circuit compilation for hybrid mpc via compiler intermediate representation. In *SECRYPT*, 2021.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[EASK14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. GraMi: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 2014.

[Ebr15] Ebrahim M. Songhori and Siam Umar Hussain and Ahmad-Reza Sadeghi and T. Schneider and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P*, 2015.

[FBL+22] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. CostCO: An automatic cost modeling framework for secure multi-party computation. In *EuroS&P*, 2022.

[FHK+14] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C compiler for secure two-party computations. In *Compiler Construction*, 2014.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure simd circuits. In *CRYPTO*, 2015.

[GMW87] Oded Goldreich, Silvio M. Micali, and Avi Wigderson. How to play any mental game. In *STOC*, 1987.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 2000.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *S&P*, 2019.

[HKoS+10] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for automating secure two-party computations. In *CCS*, 2010.

[HLOWI16] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. High-precision secure computation of satellite collision probabilities. In *SCN*, 2016.

[HS13] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *CCS*, 2013.

[HST+21] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. LLVM-based circuit compilation for practical secure computation. In *ACNS*, 2021.

[IMZ19] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *CCS*, 2019.

[JR18] Magnus Jacobsson and Peter Ryszkiewicz. A web application for interactive visual editing of graphviz graphs described in the dot language.

https://github.com/magjac/graphviz-visual-editor, 2018.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.

[Ker11] Florian Kerschbaum. Automatically optimizing secure computation. In *CCS*, 2011.

[Ker13] Florian Kerschbaum. Expression rewriting for optimizing secure computation. In *CODASPY*, 2013.

[KMSB13] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security*, 2013.

[KNR+17] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. In *CCS*, 2017.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure mpc with dishonest majority. In *CCS*, 2013.

[LA04] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, 2004.

[LSI+23] Benjamin Levy, Ben Sherman, Muhammad Ishaq, Lindsey Kennard, Ana Milanova, and Vassilis Zikas. Compilation and backend-independent vectorization for multi-party computation. Cryptology ePrint Archive, Paper 2023/089, 2023.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S&P*, 2015.

[LYK+19] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. HoneyBadgerMPC and AsynchroMix: Practical asynchronous MPC and its application to anonymous communication. In *CCS*, 2019.

[Mal11] Lior Malka. VMCrypt: Modular software architecture for scalable secure computation. In *CCS*, 2011.

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P*, 2016.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - a secure two-party computation system. In *USENIX Security*, 2004.

[MPT20] Ciaran McCreesh, Patrick Prosser, and James Trimble. The glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In *Graph Transformation*, 2020.

[MR18] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *CCS*, 2018.

[MSY21] Jean-Pierre Münch, Thomas Schneider, and Hossein Yalame. VASA: Vector AES instructions for security applications. In *ACSAC*, 2021.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *S&P*, 2017.

[NWI+15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. In *S&P*, 2015.

[OBW20] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Unifying compilers for SNARKs, SMT, and more. Cryptology ePrint Archive, Paper 2020/1586, 2020.

[OBW22] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *S&P*, 2022.

[PSSY21] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. SynCirc: Efficient synthesis of depth-optimized circuits for secure computation. In *HOST*, 2021.

[RCC+22] Neal Richardson, Ian Cook, Nic Crane, Dewey Dunnington, Romain François, Jonathan Keane, Dragoș Moldovan-Grünfeld, Jeroen Ooms, and Apache Arrow. arrow: Integration to apache arrow. https://arrow.apache.org/docs/r/, 2022.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *S&P*, 2014.

[RJHK19] M. Sadegh Riazi, Mojan Javaheripi, Siam U. Hussain, and Farinaz Koushanfar. MPCircuits: Optimized circuit generation for secure multi-party computation. In *HOST*, 2019.

[RK07] Steve Reinhardt and George Karypis. A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph. In *IEEE International Parallel and Distributed Processing Symposium*, 2007.

[RSH19] Aseem Rastogi, Nikhil Swamy, and Michael W. Hicks. Wys*: A DSL for verified secure multi-party computations. In *POST*, 2019.

[Sch18] Berry Schoenmakers. MPyC: Multiparty computation in python. https://www.win.tue.nl/~berry/mpyc/, 2018.

[SKM11] Axel Schropfer, Florian Kerschbaum, and Gunter Muller. L1 - an intermediate language for mixed-protocol secure computation. In *COMPSAC*, 2011.

[Sol10] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 2010.

[Sol19] Christine Solnon. Experimental evaluation of subgraph isomorphism solvers. In *Graph-Based Representations in Pattern Recognition*, 2019.

[ST19] Nigel Smart and Stefan Tillich. (Bristol Format) circuits of basic functions suitable for MPC and FHE. https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html, 2019.

[SZ13] Thomas Schneider and Michael Zohner. GMW vs. Yao? efficient secure two-party computation with low depth circuits. In *FC*, 2013.

[TFS+15] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *SOSP*, 2015.

[TZ16] Nilothpal Talukder and Mohammed J. Zaki. A distributed approach for graph mining in massive networks. *Data Min. Knowl. Discov.*, 2016.

[V+] Kenton Varda et al. Cap'n Proto. https://capnproto.org/index.html.

[vO14] Wouter van Oortmerssen. FlatBuffers. https://github.com/google/flatbuffers, 2014.

[vO17] Wouter van Oortmerssen. FlexBuffers. https://google.github.io/flatbuffers/flexbuffers.html, 2017.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMPtoolkit: Efficient multiparty computation toolkit. https://github.com/emp-toolkit, 2016.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[ZDC+21] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. Cerebro: A platform for Multi-Party cryptographic collaborative learning. In *USENIX Security*, 2021.

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. *Cryptology ePrint Archive, Paper 2015/1153*, 2015.

A USABILITY FUSE BACKENDS

Plaintext Interpreter. To check if a functionality in FUSE IR describes the correct semantics, it can be tested by evaluation on plaintext values. Our plaintext interpreter backend evaluates the functionality on given input values and returns the results of the computation. Given the input values for all input nodes of the circuit, the circuit is evaluated node-by-node in topological order. The result of this computation is stored in a map that maps node identifiers to each node's output values. We refer to this special mapping as the evaluation environment of the circuit. These environments are also used to evaluate calls: Whenever a call node occurs, a new environment is created for the callee circuit and the callee is evaluated under this new environment. In the end, the mappings for all the output nodes are gathered and yield the output value of the circuit.

DOT Code Generation for Plotting Circuits. As FUSE IR is built as a binary format, it is not human-readable by design. However, to introduce some form of human-readability, we have implemented a backend for the Graphviz DOT [GN00] language that can be fed into tools like Graphviz to generate a visual representation of FUSE IR. Additionally, paired with a FUSE frontend, this enables translating a language without a human-readable export function to FUSE IR to look at the functionality in a human-readable way. This way, even formats that are no longer maintained actively like HyCC [BDK+18] can get a human-readable representation through FUSE IR. In addition to viewing FUSE IR via the DOT format, it would be possible to implement a UI for editing FUSE IR by extending a DOT format editor such as [JR18] and re-importing the result via a FUSE frontend for DOT format. We leave such an extension as future work.

B ADDITIONAL RELATED WORK

B.1 Intermediate Representations

In the domain of general-purpose programming languages, the widely known LLVM project [LA04] provides the LLVM Intermediate Representation (LLVM IR) which is a generalized Static Single-Assignment (SSA) assembler language with a rich instruction set. LLVM decouples high-level programming languages from the target CPU architecture and was mainly designed for the C/C++ programming languages, but there are multiple frontends available, including Haskell, Go, and Rust. LLVM provides a rich toolchain to implement and execute analysis and optimization passes on LLVM IR, which depend on the SSA form. In [HST+21], the authors have used LLVM to automate circuit compilation for MPC with an optimizer suite the produce optimized circuits. However, the problem with using LLVM IR as an IR for MPC is that the language is too expressive, such that the generated intermediate representation from high-level code like C/C++ can hardly be used directly in MPC contexts. Similarly to LLVM IR, [DKS+21] presents a graph-based IR for MPC that is implemented as a new stage between HyCC [BDK+18] circuits and the optimization phase of HyCC. The main difference between FUSE and [HST+21, DKS+21] is that FUSE does not only design an IR for MPC, but also a file format and a rich toolchain to operate on the format.

Open Neural Network Exchange (ONNX) Standard [BLZ+19] represents machine learning models and is used in the MPC framework MOTION2NX [BCS21]. It also defines an underlying file format which is implemented using Google Protocol Buffers.[6] As opposed to FlatBuffers, Protocol Buffers always need to be unpacked before usage, making them less memory-efficient.

In the domain of Zero Knowledge (ZK) proofs, the CirC compiler infrastructure [OBW20, OBW22] implements a shared infrastructure for compiling to special constraints in the form of existentially quantified circuits. These are used to generate both proofs for Satisfiability Modulo Theories (SMT) and ZK. The computation format CirC-IR describes stateless, non-deterministic, and non-uniform computation. However, it has a more abstract view on functionalities, such that the gap between CirC-IR and MPC targets is larger than what we aim for in FUSE. However, similar to the case of LLVM [LA04], we could benefit from this work by providing translations to and from FUSE IR.

B.2 Serialization Frameworks

Google Protocol Buffers[6] (protobuf) is very similar to FlatBuffers (§2.3) and used in numerous projects, such as ONNX [BLZ+19] and envoy proxy[7]. It also uses schemas to define the data types inside the serialized binary and both frameworks do not incur extra space for optional fields that have not been assigned a value. Every field in a protobuf schema must have a unique identifier that cannot be changed once set, making schema description and evolution more complicated for users. Protobufs must be unpacked before access, even if the access is read-only.

Another schema-based serialization framework with support for schema evolution and zero-copy reads is Cap'n Proto [V+]. It was built to be used in Sandstorm[8] and is used in Cloudflare Workers[9]. When serializing data with Cap'n Proto, the order in which the parts of the data is serialized does not matter, which is the case for FlatBuffers. However, fields that are not set do impose an additional memory-overhead, making the generated binaries with Cap'n Proto larger than with FlatBuffers if there are unused fields.

Apache Arrow [RCC+22] is a columnar memory format for both flat and hierarchical data that is specially organized for efficient analytical operations on modern hardware. It also has numerous use cases, such as Apache Spark, the FPGA-framework Fletcher[10], and the pandas[11] data analysis framework. Because it uses FlatBuffers [vO14] under the hood, it also provides read-access to serialized data without unpacking. It provides more mechanisms built on top of FlatBuffers like null values to abstract over values that are not present in the FlatBuffer's virtual table. Apache Arrow does not fix a schema and adhere to it, but generates a FlatBuffer schema on the fly for the serialized data to ensure fast serialization and zero-copy reads. These techniques are likely to introduce an additional memory-overhead for the generated binaries which is why we have decided to directly work with FlatBuffers for the choice of our serialization framework. However, unlike FlatBuffers, Apache Arrows provides data streaming out of the box and was built specifically with large-scale data processing in mind.

[6] https://developers.google.com/protocol-buffers
[7] https://github.com/envoyproxy/envoy
[8] https://github.com/sandstorm-io/sandstorm
[9] https://developers.cloudflare.com/workers/
[10] https://github.com/abs-tudelft/fletcher
[11] https://github.com/pandas-dev/pandas

C INSTRUCTION VECTORIZATION

Our Algorithm. The instruction depth of gate g for operation o is the maximum number of o-gates from any input gate to g. We first group gates of a particular operation o by their instruction depth to identify suitable candidate lists for vectorization. This method of candidate selection ensures that no output-input relation exists between two vectorized gates, as such gates could not be executed in parallel. Indeed, if two o-gates have the same instruction depth for o, they cannot have an output-input relationship. Further, we consider only candidate lists with equal to or more than `Min_Gates` entries for vectorization because it might not be beneficial to vectorize fewer than `Min_Gates` gates.

The general depth of gate g is the maximum number of gates from any input gate to g. We use the general depth to estimate how far two gates are apart in each candidate list, and exclude distant candidates, which would cause a significant delay when executed in parallel. Indeed, we calculate the general depth median of each list, and prune gates with a difference to the median greater than `Max_Distance`. Finally, the candidate lists are transformed into SIMD gates.

FUSE offers Instruction Vectorization for one specific operation and for all operations simultaneously. It generates a report that contains the total number of replaced gates per operation. The parameters `Min_Gates` and `Max_Distance` can be set by users of the optimization pass.

D FREQUENT SUBCIRCUIT ANALYSIS COMPUTATION STAGES

Stage 1 - Frequent Subgraph Mining. Frequent Subcircuit Replacement is a two-stage process that first identifies patterns, i.e., the same instruction sequence at different positions in the circuit, and subsequently replaces each occurrence by a call node. For the first stage, we transform the MPC circuit into a graph format and apply an algorithm from the domain of Frequent Subgraph Mining, which enumerates all subgraphs that repeatedly occur with a frequency above some minimum threshold. Every node in the graph corresponds to a node in the circuit, and every edge corresponds to a successor relationship between two nodes.

DistGraph [TZ16] is a state-of-the-art Frequent Subgraph Mining algorithm that supports sequential, thread-parallel and distributed modes of operation. It can operate on graphs that would otherwise be too large to fit in the memory of a single compute node by partitioning the input graph into different segments. In this context, DistGraph can handle massive networks with over a billion vertices and four billion edges [TZ16], and offers a scalable and flexible solution for the first stage of our optimization. Different Frequent Subgraph Mining algorithms [EASK14, RK07, TFS+15] further restrict the modes of operation or assume that the input graph fits into the memory of a single compute node. FUSE currently supports sequential DistGraph execution, which does not segment the input graph. The runtime of DistGraph, the size and the amount of identified patterns is highly dependent on the user-provided parameter `Min_Frequency`, i.e., the minimum frequency of a subgraph to be considered by the algorithm. A larger value generally implies smaller, identified patterns and a smaller DistGraph runtime because the search space can be efficiently pruned during the calculation. We found in our experiments that even for small fluctuations in the parameter value, the runtime and the output might change significantly – resulting in infeasible runtimes or tiny patterns. Thus, it is essential to simplify the parameter selection of a 'good' `Min_Frequency` in the FUSE framework. DistGraph is supposed to identify frequent patterns that, when replaced, improve the memory footprint of the circuit. Surprisingly, we found that for large patterns, the replacement often produces circuits with an increased memory-footprint compared to the original circuit. This is most likely due to the overhead of the newly created subcircuit, and the fewer occurrences of larger patterns: A minimum amount of calls is required to compensate the overhead. Small patterns also increase the memory-footprint because call nodes are more memory consuming than standard gates. During experiments, we found that both too large and too small patterns do not improve memory efficiency of the circuit, thus, we want to identify and replace medium-sized patterns to achieve an improvement in memory consumption.

We provide an automatic time-based parameter search that searches for an optimal `Min_Frequency` $\in [2, \texttt{numGates}]$ via binary search, where `numGates` is the number of gates in the MPC circuit. We observed that DistGraph terminates within a small timeout for parameters that produce small- and medium-sized patterns. Thus, our parameter search returns a parameter, s.t. DistGraph terminates in a feasible time, and finds medium-sized patterns. Furthermore, we provide the option to further post-process the DistGraph output to include the largest (`Mode=0`), the second largest (`Mode=1`), or the third largest (`Mode=2`) patterns. In some cases, this produces different results with respect to the number and locations of replacements. The automatic parameter search for `Min_Frequency` can also search for the best `Mode` parameter (`Try_Modes>1`).

Stage 2 - Finding Subgraph Isomorphisms. The first stage of the Frequent Subcircuit Replacement optimization identifies frequent patterns in the circuit. In the second stage, we create a subroutine for every pattern and replace its embeddings in the circuit by subroutine calls. To identify the embeddings, we apply an algorithm for finding Subgraph Isomorphisms, i.e., finding a small pattern graph inside a larger target graph or determining its non-existence. The result is a set of mappings from the vertices of the pattern graph to the vertices of the target graph [MPT20]. Subgraph Isomorphism algorithms can be classified into the following categories: backtracking- and connectivity-based algorithms [BG17, CFSV18], and those based upon constraint programming [Sol10, MPT20].

While the constraint programming-based algorithms provide better performance on hard instances, backtrackers will generally run faster and with less memory footprint on easy instances due to lower overheads and faster startup costs [Sol19, MPT20]. Constraint programming adds additional implied constraints that can speed up the solving process, such as filtering nodes by their neighborhood before considering an embedding [MPT20]. The Glasgow Subgraph Solver [MPT20] uses a constraint programming-based approach and performs best on hard instances [Sol19]. We use the sequential algorithms provided by the Glasgow Subgraph Solver to gracefully cover hard instances in our optimization.

We subsequently filter the output of the Glasgow Solver to exclude invalid embeddings before the replacement. (1) Overlapping

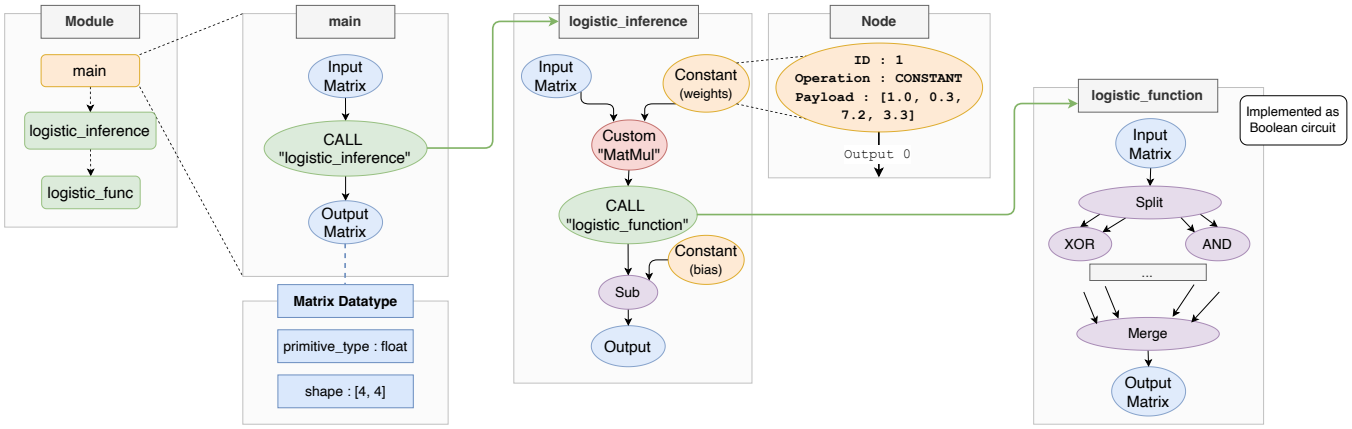

Figure 3: Example Module in FUSE IR for Logistic Regression Inference.

Embeddings: Two embeddings contain the same node, which can only be replaced once by a subroutine call. (2) Cyclic Embeddings: A cyclic embedding contains outputs that feed into the inputs of the subcircuit. However, during a call to a subcircuit, all input values need to be concrete, which is not the case for such circular dependencies. We perform a pruned depth-first search at every output node to identify and exclude circular embeddings. FUSE generates a report for the Frequent Subcircuit Replacement optimization that summarizes the number of replaced embeddings for each pattern.

E FUSE IR EXAMPLE: LOGISTIC REGRESSION

In Figure 3 we present an example FUSE IR module that describes Logistic Regression inference for an already trained Logistic Regression model. The module contains three functions, of which the `main` function marks the entry point for execution. The `main` function takes in a 4×4 float matrix, calls the `logistic_inference` function and returns a float matrix of the same dimensions. In `logistic_inference`, the weights and bias matrices are constants, since we initially assumed that the model was trained already. In our example, the weights vector contains the values `[1.0, 0.3, 7.2, 3.3]` which are stored inside the `payload` field of the node.

Since matrix multiplication has special protocols, e.g., in [MR18, MZ17], we can describe this with a custom node with the operation name `MatMul`. The output of this matrix multiplication is given to the `logistic_function`, which in this example is implemented as a Boolean circuit. This assumption is justified, since in MPC non-linear function are often more efficient in the Boolean than in the Arithmetic world. To do that, the matrix values are split into Boolean wires and in the end merged together to produce the output matrix.