FLUENT: A Tool for Efficient Mixed-Protocol Semi-Private Function Evaluation*

Daniel Günther, Joachim Schmidt, Thomas Schneider, and Hossein Yalame
Technical University of Darmstadt, Darmstadt, Germany
Email: {guenther, schneider, yalame}@encrypto.cs.tu-darmstadt.de, joachim.schmidt@stud.tu-darmstadt.de

***Abstract*—In modern business-to-customer interactions, handling private or confidential data is essential. Private Function Evaluation (PFE) protocols ensure the privacy of both the customers' input data and the business' function evaluated on it, which is often sensitive intellectual property (IP). However, fully hiding the function in PFE results in high-performance overhead. Semi-Private Function Evaluation (SPFE) is a generalization of PFE to only partially hide the function, whereas specific non-critical components remain public. Our paper introduces a novel framework designed to make SPFE accessible to non-experts and practical for real-world deployments.**

To achieve this, we improve on previous SPFE solutions in two aspects. First, we enhance the developer experience by leveraging High-Level Synthesis (HLS), making our tool more user-friendly than previous SPFE frameworks. Second, we achieve a $2\times$ speedup compared to the previous state-of-the-art through more efficient underlying constructions and the usage of Lookup Tables (LUTs).

We evaluate the performance of our framework in terms of communication and runtime efficiency. Our final implementation is available as an open-source project, aiming to bridge the gap between advanced cryptographic protocols and their practical application in industry scenarios.

***Index Terms*—Multi-Party Computation, Privacy, Private Function Evaluation**

I. Introduction

In today's digital landscape, maintaining the confidentiality of processes that handle private data is essential for both economic and security reasons. Typically, these processes are executed on servers rather than local devices, which requires end users to trust that these services will manage their data responsibly and ensure its deletion after processing. However, numerous data leaks have demonstrated that this trust is often not justified, with breaches impacting companies across various sectors [2], [3].

A prominent example of the need for data protection measures is hardware intellectual property (IP) protection. Circuit designs are critical assets for companies as they contain significant intellectual property. Manufacturers who produce these designs have the potential to access and learn these proprietary details, leading to IP theft and unauthorized cloning, which can result in substantial financial losses and diminished competitive edge [4].

Hardware logic locking [5], [6] has emerged as a potential solution for hardware IP protection. This technique involves embedding a locking mechanism within the hardware to prevent unauthorized use. However, current methods of logic locking are not fully secure. Attackers can bypass or reverse-engineer these protections, which compromises the security of the designs [3].

Private Function Evaluation (PFE) [7] provides a technical solution that protects both data and functions and is suitable for applications like hardware IP protection. PFE is a generalization of Secure Function Evaluation (SFE), a cryptographic protocol that allows N parties with private inputs $x_1, \ldots, x_N$ to securely compute a public known function f on their private inputs and obtain nothing but the result $f(x_1, \ldots, x_N)$. In PFE, one of the parties, denoted as P_1, provides a private function f, and the other parties provide private input data $x_1, \ldots, x_N$. At the end of the protocol, all parties securely compute $f(x_1, \ldots, x_N)$ while learning no additional information about the inputs of other parties or the function (except an upper bound on its size).

However, hiding the structure of the function in PFE comes at a high cost: Today's most flexible technique for PFE is based on Universal Circuits (UCs) [8], Boolean circuits programmable to compute any function consisting of n gates, which yield inevitable complexity of $\Theta(n \log n)$ [9]. In real-world applications, the function f often contains non-critical components that need not be kept confidential (e.g., it is natural to run comparisons on an age input). Revealing these to the users would still maintain privacy while allowing for much better efficiency. Functions consisting of private and public components are called *semi-private* and their secure evaluation is called Semi-Private Function Evaluation (SPFE) [10].

There are two main implementations of SPFE: FairplaySPF [10] and the CBMC-GC based framework of [11], both briefly described in §V. A common disadvantage of these frameworks is their limited usability. They rely on a domain-specific language and require a low-level understanding of Boolean circuit designs. Moreover, in the CBMC-GC based framework, complete refactoring of the function programming is required when the implementation of a single function's component changes. Therefore, a significant gap exists for an efficient SPFE tool that can completely hide the private function components while remaining accessible to developers.

To fill this gap, we introduce FLUENT, an efficient and user-friendly SPFE tool. Unlike FairplaySPF [10] and the framework of [11], FLUENT simplifies the implementation of semi-private functions in the high-level languages C and C++ and allows for the first time to combine the private and public function components easily. A developer can directly

[*] Please cite the conference version published at ACSAC'24 [1].

mark a sub-function as public or private in the high-level program. Additionally, FLUENT allows implementing semi-private functions in the low-level hardware description language Verilog. FLUENT is *the first* SPFE tool that compiles from an existing programming language instead of a new domain-specific language.

A. Practical Applications of SPFE

Hardware design processes typically involve multiple parties such as IP vendors, toolchain developers, and hardware designers. Each contributes proprietary products to the development pipeline, which are trade secrets. The collaboration in this industry thus relies on trust, which is hard to earn for startups that are not established in their respective space. One approach to fixing this issue was proposed by Hashemi et al. [12] with their Garbled Electronic Design Automation (EDA) framework. Using PFE, Garbled EDA allows hardware designers to evaluate and test their proprietary designs privately while toolchain and IP vendors reveal minimal information about their IP. The main example in [12] is a Verilog hardware module simulation. More formally, the hardware IP is the function f evaluated on private inputs x. FLUENT also supports this use case.

Other applications of PFE include *privacy-preserving credit checking* [13]. Here, a financial institution can check the suitability of a credit for a certain customer while respecting the privacy of the customer who provides their sensitive financial information. At the same time, the specific ruleset governing the credit score calculation is a trade secret and should, therefore, be kept hidden. Further (S)PFE use cases from the literature include, but are not limited to, medical diagnosis [14], stream filtering [15], private deep learning accelerators [16], and remote software diagnosis [17].

B. Our Contributions

For *the first time*, FLUENT provides a user-friendly tool that integrates PFE, SFE, and SPFE, along with support for LUT-based circuits, enabling secure evaluation within an SFE framework. We discuss related work in more detail in §V. Our main contributions are as follows:

1. ***A Novel Tool for LUT-based SPFE.*** We develop and implement FLUENT, a new accessible and comprehensive SPFE tool. It allows to easily implement semi-private functions in the high-level languages C/C++ and the low-level hardware description language Verilog. FLUENT allows developers to mark a sub-function as public or private directly in the high-level program, using pragmas for C/C++ or module attributes in Verilog, eliminating the need for refactoring the source code after small function changes as in [11]. Moreover, in contrast to [10], [11], our tool is designed to be easily extendable and allows to compile sub-functions into Lookup Table (LUT)-based circuits. Private sub-functions are evaluated with UCs by integrating recent optimizations for LUT evaluation within UCs [18]. However, that UC framework exclusively supports complete private circuits, while FLUENT handles private sub-circuits, making our tool the first to implement semi-private functions with LUTs.

On top of that, FLUENT is not only suited to implement semi-private functions for SPFE, but can also be used to fully hide functions for PFE, providing a user-friendly tool for developing private functions. Finally, our SPFE tool outputs a circuit description file in a format for which we have integrated support within the ABY [19] framework for SFE. We provide an open source implementation of FLUENT.[1] Leveraging existing conversion capabilities of the ABY framework [19], we allow users to use different protocols for certain parts of an application to exploit the respective advantages of each. Public modules can be evaluated with Yao's Garbled Circuits protocol [20] or with the SP-LUT protocol [21].

2. ***Benchmarks and Applications.*** We benchmark the performance of FLUENT and demonstrate its practical relevance by measuring the overhead of hiding basic and arithmetic operations including floating point operations ($51.3\times - 68.4\times$ overhead for hiding these operations) and SHA256 calculations ($129.4\times$ overhead for hiding). FLUENT provides a straightforward solution to avoid this overhead by making certain parts of the computation public, which can significantly reduce overhead based on the specific application's requirements. Taking the example of a car insurance tariff calculation function, where only critical sub-functions are hidden while non-critical operations remain public, FLUENT outperforms the framework of [11] by $2\times$ in runtime, and over full PFE (the whole function is hidden) by $7.7\times$.

C. High-level Overview of Our Tool

In Fig. 1 we give an overview over the structure of our tool FLUENT. The displayed process is controlled by a novel compiler driver which is also responsible for merging different components (cf. §III-C). In addition to implementing the compiler driver, various components used in the compilation process were modified to be compatible with our workflow.

Users of FLUENT can decide whether to write their desired semi-private function in C/C++ or Verilog; the compiler automatically performs an additional HLS pass using Google XLS [22] if C/C++ source code is provided. In contrast to the design of individual functions as in [11], this approach removes the burden of manually compiling and integrating all components.

When using C/C++ source files, some restrictions apply. Dynamic memory allocation with `malloc` calls or the `new` operator is disallowed. Furthermore, any control flow not statically known at compile-time is not synthesizable. Among these is `setjmp`/`longjmp`. Apart from being unsupported by XLS, these constructs cannot be mapped to combinational Boolean circuits. Additionally, loops must have a static bound and be unrolled using a pragma. The main reason for this is that private modules shall only leak an upper bound on the function size. If loops are not unrolled and the same code is computed multiple times, this would leak the existence of a loop and how often the loop has been run. In contrast, public modules do not necessarily have this requirement. Nevertheless, we aim to enforce a coding style where developers can change module

[1] Our code is published under MIT license at: https://encrypto.de/code/FLUENT.

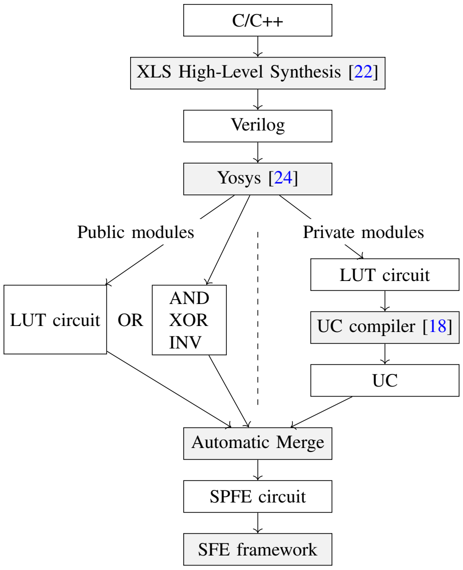

Fig. 1: High-level comparison of the compiler pipeline of our tool FLUENT.

visibilities from public to private by only changing a single line of code without adapting loops. Therefore, we also require loop unrolling in public modules.

While private modules are always compiled to k_{priv}-input LUT circuits, our tool allows the user to choose the compilation mode of public modules. Public modules can be lowered to Boolean circuits composed of AND, XOR, and INV gates for evaluation with Yao's garbled circuits protocol [20], [23]. Alternatively, they can be compiled to k_{pub}-input LUT circuits for evaluation using the SP-LUT protocol [21]. The output SPFE circuit description file combines all modules, public and private, in a single file for secure evaluation in a SFE framework like ABY [19].

II. Preliminaries

Universal Circuits (UCs). Universal Circuits (UCs) [8] are special Boolean circuits of size $\Theta(n \log n)$ [9] that can be programmed to compute any Boolean function $f(x)$ of at most n gates on input x. More formally, a universal circuit UC takes programming bits p_f describing the functionality of the function f and the function's input x as inputs and computes $UC(p_f, x) = f(x)$. The idea and first two constructions of UCs were provided by Valiant [8]. Today, these two constructions are known as 2-way and 4-way split constructions referencing their recursive structure. These UC constructions have sizes of $\sim 5n \log n$ and $\sim 4.75n \log n$, respectively.

As Valiant already provided UC constructions of asymptotically optimal size $\Theta(n \log n)$, UC research mostly focused on reducing the concrete size through optimizations [7], [25], [26], [27] and the prefactor of the superlinear term of the size [28], [29]. The state-of-the-art UC construction by Liu et al. [29] removes redundancies of Valiant's UC constructions and achieves an asymptotic size of $\sim 3n \log n$.

All these works have in common that the UC is designed to evaluate Boolean circuits consisting of 2-input and 1-output gates. Recently, [18] extended the UC construction of Liu et al. [29] to evaluate ρ-input and ω-output LUTs. They propose two different constructions, both of which have advantages and disadvantages.

Their first construction is called *LUC* and has asymptotic size $\sim 1.5\rho\omega n \log \omega n$. While this construction can hide a function fully, it unfortunately has the LUT's input dimension ρ as a prefactor. Many functions can be most efficiently represented as multi-input LUTs of various input dimensions, e.g., AES SBoxes can naturally be implemented with 8-input LUTs. However, mathematical operations like additions and multiplications consist of full adders and thus benefit from using 3-input LUTs. Combining a circuit that uses AES and arithmetic operations and evaluating it with the LUC construction brings a massive overhead as all 3-input LUTs need to be extended into an 8-input LUT, i.e., the potential of many 8-input LUTs is not fully used, but its expensive prefactor is omnipresent. For this reason, Disser et al. [18] proposed a second construction called *VUC* that has asymptotic size $\sim 1.5(\omega n + \Delta) \log(\omega n + \Delta)$ with $\Delta = \sum_{i=1}^{n} \frac{L_i^{in} - 2}{2}$, where L_i^{in} is the number of inputs of the i-th LUT of the circuit. Here, the actual size of the VUC constructions depends on the used LUTs's dimensions. However, this construction leaks the number of inputs of each individual LUT and thus leaks more information than the LUC construction.

Secure Function Evaluation (SFE). Secure Function Evaluation (SFE) allows N parties to compute a public known function $f(x_1, \ldots, x_N)$ on their respective private input data $x_1, \ldots, x_N$ without learning anything but the output. The function f is commonly implemented as a Boolean circuit consisting of AND and XOR gates, where XOR gates can be evaluated without interaction [30], [31], whereas AND gates require communication between the parties.

SFE protocols based on secret sharing [30], [32], [33], [34], [35], [36], [37] require communication rounds that are linear in the multiplicative depth of the circuit, but are well-suited for high-throughput applications due to their low amount of communication. Yao-based protocols [20] have a constant number of rounds and are well-suited for low-latency solutions. The mutually orthogonal goals of these two approaches paved the way for LUT-based SFE, which provides a balance between total communication and online round complexity [21], [36]. An effective implementation of LUT-based SFE was introduced under the name SP-LUT in [21]. To further enhance efficiency, an alternative approach involves the initial transformation of the LUT representation into a Boolean expression, which is evaluated with a multi-fan-in inner product protocol, as proposed in FLUTE [36].

Yao's Garbled Circuits [20] allows two parties to securely evaluate a public function $f(x, y)$ on their respective private inputs x and y. Today's most efficient solution for garbled circuits combines point-and-permute [38], free-XOR [31], fixed-key AES [39], and three-halves-garbling [40]. With these optimizations, each AND gate requires communication 1.5κ bits for security parameter κ in the setup phase and XOR gates have no communication. In the context of PFE on UCs, a constant number of communication rounds is preferable as UCs have a depth of $\mathcal{O}(n)$, where n is the size of the function f, and thus would require $\mathcal{O}(n)$ communication rounds [9].

LUT-based Circuit Synthesis. There exists a spectrum of commercial FPGA synthesis tools such as Intel HLS Compiler [41], Microchip SmartHLS [42], and AMD Xilinx Vivado [43]. However, these tools synthesize LUT-based circuits that are tailored to their devices' specifics, such as the size and quantity of physically available LUTs. In FLUENT, we generate up to 8 input LUTs, which is not possible with commonly used commercial tools. The Berkeley Logic Synthesis tool ABC [44] allows to map circuits to variable input LUTs using a technology named priority cuts [45]. For both standard gates and LUTs, ABC offers an experimental implementation of various mapping and optimization techniques based on optimal-delay directed acyclic graph (DAG) technology mapping. As ABC allows the user to choose the maximum number of LUT inputs, regardless of any target-specific FPGA design characteristics, it is a great fit for our needs. We combine the mapping of ABC [44] with Yosys [24] as follows: ABC [44]: The Boolean circuit network is first organized into a directed acyclic graph (DAG) with 2-input, 1-output nodes, and then this graph is mapped into multi-input LUTs via the Yosys-ABC toolchain [24], [44]. Following [21], [36], the maximum number of inputs to our LUTs is restricted to 8, as this offers a good balance between performance and efficiency.

More concretely, in FLUENT, private modules are compiled to k_{priv}-input LUT circuits which can be embedded into UCs. Public modules are decomposed into multiple modules and then each module is translated into two different circuit formats, i.e., LUTs and Boolean gates. We utilize the JSON [46] backend for public circuits as JSON is a widespread format and JSON parsing is simpler than writing a parser for another output format by hand. The state-of-the-art UC compiler of [18] only supports the Bench [47] format, which can be emitted by ABC. We describe the exact usage of Yosys and ABC in §III-D.

High Level Synthesis Tools. Traditionally, hardware development involved behavioral and structural descriptions of the desired circuit in hardware descrpition languages (HDLs) such as Verilog or VHDL. Since the late 1990s, High-Level Synthesis (HLS) tools were developed to bridge the gap between high-level programming languages and HDLs. One of the first was Handel-C [48], released in 1996. These tools allow users to write code in a high-level language and automatically convert it into an equivalent hardware circuit description, which in turn can be synthesized by classic logic synthesis. This makes it easier for software developers to work with hardware, as they can use their existing knowledge of high-level programming languages rather than having to learn a specialized HDL. As our intended use of HLS differs from the conventional hardware development, we have specific requirements which rule out some existing HLS tools. We require the output HDL to preserve the structure of the input code so that different functions will be lowered to different modules instead of being flattened to one module. Merging functions of the same type at this stage is possible and could be explored as a future optimization, however, we decided against this in FLUENT to maintain simplicity and ease of debugging. Furthermore, we expect the output to describe a combinational Boolean circuit which computes the desired function immediately instead of a pipelined circuit which computes the result in several clock cycles.

From the several available commercial and academic HLS tools [41], [42], [43], we selected Google's XLS toolchain [22] for compiling C/C++ code to Verilog as this tool fulfilled our requirements in addition to being open-source. The high-level synthesis pipeline in XLS operates on an intermediate representation (IR) which can either be generated from the DSLX domain-specific language or generated from C/C++ code. The IR is subsequently optimized in multiple passes and finally lowered to Verilog. Alternatively, XLS supports interpreting IR code or generating native binaries from IR with the help of the LLVM compiler infrastructure [49]. We provide an example of HLS in §A.

Listing 1: Example Verilog code for privately comparing with a threshold and publicly incrementing a counter.

```verilog
 (* private *)
module compare ( input  wire [31:0] a,
    output wire b);
  assign b = a > 4096;
endmodule
 (* public *)  // optional
module increment (input  wire [31:0] a,
    output wire [31:0] c);
  assign c = a + 1;
endmodule
module count_over_threshold (input  wire
    [31:0] value, input  wire [31:0]
    counter, output wire [31:0] new_counter
    );
  wire over_threshold;
  wire [31:0] incremented;
  compare cmp (value, over_threshold);
  increment inc (counter, incremented);
  assign new_counter = over_threshold ?
      counter : incremented;
endmodule  // main
```

III. Our FLUENT Tool

In this section, we describe our FLUENT tool for Semi-Private Function Evaluation (SPFE) that compiles a function description in C/C++ or Verilog into a circuit for evaluation in an SFE framework.

A. FLUENT in Verilog

Our FLUENT tool can interpret programs written in the hardware description language Verilog. A semi-private function consists of private and public modules, where the private modules of the function are only known to one party, and the public parts are known to all parties. Our format allows directly declaring the private modules inside the source code by adding a private attribute to the code part. For the first time, the whole semi-private function, including its privacy requirements, can be completely described in the Verilog language.

List. 1 depicts a Verilog source code example of a semi-private function for SPFE that takes a 32-bit value input and a 32-bit counter input. When the value is over a certain threshold, the incremented counter value is returned, otherwise, the unmodified counter value is returned. We designate the comparator module as private, while the increment module is public.

The Verilog language supports adding *attributes* to modules. These attributes have names and can have constant data attached to them, such as integers or strings. FLUENT uses module attributes to distinguish between public and private modules. For instance, in our example, the `compare` module has the `private` attribute attached to it, as shown in line 1. By default, a module will be considered as public, and the `public` attribute is optional, as seen in line 5. Note that our input language is standard Verilog code, i.e., our semi-private functions can be generated and processed with existing Verilog tools, including industry-grade synthesis tools such as AMD Xilinx Vivado [43].

B. FLUENT in C/C++

FLUENT can process C/C++ in addition to Verilog source files. This is achieved using the Google XLS framework [22] for an HLS pass, compiling C/C++ sources into Verilog. For this reason, the C/C++ features supported by Google XLS are also supported by FLUENT. List. 2 shows C/C++ source code for the same semi-private function as the Verilog code in List. 1.

Listing 2: Example C code for privately comparing with a threshold and publicly incrementing a counter.

```c
#pragma spfe_private
int compare(int b) {
  return b > 4096;
}

int increment(int a) {
  return a + 1;
}

int count_over_threshold(int value, int counter) {
  return compare(value) ? counter : increment(counter);
}
```

Benefits of using C/C++. Imperative programming languages like C/C++ are more familiar to most programmers than a hardware description language like Verilog. Existing codebases written in C or C++ can be reused with minimal changes, provided they do not rely on features that cannot be synthesized to circuits (see §I-C).

SystemVerilog supports structured data types, but these types are more versatile in C++ (see List. 3). Methods in C++ significantly enhance code readability when working with structured data types. While SystemVerilog supports classes with member functions, these functions do not integrate into the module hierarchy. Additionally, Yosys [24] currently does not support SystemVerilog classes unless supplemented with third-party plugins.

Listing 3: Example of methods in C++.

```cpp
struct Rectangle {
  int width;
  int height;

  int area();
};

#pragma spfe_private
int Rectangle::area() {
  return width * height;
}

int area_difference(Rectangle r1, Rectangle r2) {
  return r1.area() - r2.area();
}
```

Listing 4: Array parameters in SystemVerilog and C/C++.

```verilog
// Verilog source code
module sum(input byte x0, input byte x1, output integer out);
  integer x[2];

  always_comb begin
    x[0] = x0;
    x[1] = x1;

    out = 0;
    for (integer i = 0; i < 2; i = i + 1) begin
      out = out + x[i];
    end
  end
endmodule
```

```c
// C/C++ source code
int sum(const char x[2]) {
  int sum = 0;
#pragma hls_unroll yes
  for (int i = 0; i < 2; i++) {
    sum += x[i];
  }
  return sum;
}
```

HDL synthesis frameworks vary greatly in their supported language features. For example, Yosys forbids `while` loops in `always` blocks and does not support passing SystemVerilog unpacked arrays as parameters even though the SystemVerilog standard [50] allows them. In contrast, XLS supports array parameters, leading to simpler code when using C/C++ as demonstrated in List. 4.

Integration into FLUENT. The HLS pass has three segments:

In the first segment, the `xlscc` tool is invoked to transform the C/C++ code into the XLS intermediate representation (IR). Besides translating high-level constructs into synthesizable IR operations, functions annotated with the `spfe_private` pragma in the source code are annotated in the IR to mark that they are private modules.

The second segment optimizes the IR code with a separate tool from the XLS toolchain. Unlike a typical hardware development workflow, we deliberately skip the inlining pass, which replaces all function calls with the body of the respective function. If all functions were inlined, the Verilog code would no longer include the boundaries between functions, thus eliminating the possibility of distinguishing between private and public modules.

The third segment uses the codegen_main utility to convert the optimized IR to Verilog code. We communicate to the code generator that we intend to emit combinational instead of pipelined Verilog. If the output were pipelined, the resulting circuit would compute the function during multiple clock cycles instead of instantaneously. While pipelining is essential in hardware development as it increases throughput, neither UCs nor the SFE protocols we use in our tool support such circuits. The final output is a Verilog module hierarchy, where every function in the input C/C++ code is translated to a Verilog module, and function calls are expressed as module instantiations. An example of the HLS process is shown in §A.

C. Compiler Pipeline

Similar to traditional compilers, our FLUENT circuit compiler contains a pipeline (cf. Fig. 1) on page 3) with intermediate stages between the C/C++ input and the SPFE circuit output.

The compiler pipeline is managed and executed by the *compiler driver*. The compiler driver is a standalone executable that executes the high-level synthesis framework XLS [22], the synthesizer for the Verilog language Yosys [24], and the UC compiler [18], described in detail in §III-D. In addition, it is responsible for merging the public and the private modules of the semi-private function into one resulting SPFE circuit, described in §III-E. The resulting SPFE circuit can be compiled and evaluated with the ABY framework [19] for Secure Function Evaluation (SFE) as described in §III-G.

The compiler driver is implemented in Zig [51], a low-level general-purpose programming language inspired by C and Rust. The Zig toolchain includes a C/C++ compiler and can cross-compile to Windows, Linux, and macOS. Even though Zig is less widely adopted than the Rust programming language, it was chosen as its simplicity allowed faster development and the integrated C/C++ toolchain enabled tight integration with the existing codebases.

D. Yosys and UC compiler Invocations

Each module in the module hierarchy is processed differently in the compiler pipeline depending on the module's designated visibility. Our FLUENT tool distinguishes between public and private modules: Public modules are those parts of the function that are known to all parties in the Secure Function Evaluation (SFE) protocol, while private modules are the parts of the function only known to the party who inputs the function. While public modules can directly be processed by the underlying SFE framework, hiding the private modules requires additional steps. For these, we use a Universal Circuit (UC) for each private module that gets programmed such that it simulates the private module's functionality using the state-of-the-art UC compiler of [18].

Public Modules. FLUENT allows the user to choose whether to evaluate public modules with any of the following state-of-the-art SFE protocols: Yao's garbled circuits protocol [20] (compiled to a Boolean circuit with 2-input gates) or with the SP-LUT protocol [21] (compiled to a multi-input LUT-based circuit). When choosing Yao's garbled circuits protocol, the public modules must be compiled into Boolean circuits consisting of AND, XOR, and INV gates. Ideally, the resulting Boolean circuit should minimize the number of AND gates as XOR gates can be evaluated locally without any communication [31]. The compilation process for this set of gates involves both Yosys [24] and ABC [44]. If the user instead chooses the SP-LUT [21] protocol, Yosys is instructed to generate circuits with k_{pub}-input LUTs.

During synthesis, Yosys performs a variety of optimizations on the intermediate circuit descriptions within the opt pass. Among these is the opt_expr which performs constant folding. For instance, it simplifies expressions like $x \wedge 1$ to just x, and $x \wedge 0$ becomes 0. Scenarios where constant inputs are used include the compare function in List. 2 or SBOX LUTs found in many AES implementations like OpenSSL [52].

As all public modules are of the same type (either all evaluated with Yao's garbled circuits or all have the same number of LUT inputs k_{pub}), all private modules are marked as black box modules. These black box modules are treated as modules with known ports but unknown cells. This enables Yosys to flatten the module hierarchy, preserving only the black box modules. Flattening the hierarchy of public modules in Yosys instead of externally in our compiler driver enables Yosys to optimize across module boundaries.

Allowing various public modules to have different configurations could make flattening modules in Yosys unfeasible, as it would blur the distinction between modules suitable for marking as black boxes and those intended for flattening. Future work could explore implementing diverse public modules, each with its own optimizations.

Private Modules. To process private modules, we iterate through all modules with the private attribute attached to them. First, we compile each module in question to a k_{priv}-input LUT circuit. The resulting circuit is then compiled to a UC and the corresponding programming bits using the UC compiler of [18]. Like the process for public modules, the synthesis workflow for private modules involves Yosys [24] and ABC [44]. However, we require the synthesized circuit to be in the Bench [47] format as the UC compiler [18] does not support other Yosys or ABC output formats. Therefore, we manually invoke ABC, as Yosys does not support outputting Bench.

We invoke Yosys and convert the high-level cells to simple Boolean operations. To transfer this intermediate data to ABC, we output the processed circuit in the Berkeley Logic Interchange Format (BLIF) [53] supported by Yosys and ABC. Once loaded into ABC, we run various optimizations and map the circuit to a k_{priv}-input LUT architecture using the if field-programmable gate array (FPGA) technology mapping pass. We then output the circuit into a Bench file and invoke the UC compiler [18] to get the UC file and corresponding programming bits.

E. Merging Public and Private Modules

Once all modules in the design hierarchy have been converted to either UCs or LUT/Boolean circuits, the compiler driver needs to flatten the hierarchy into a single circuit which can be evaluated by the SFE framework ABY [19]. Although

Yosys [24] natively supports flattening of design hierarchies with the `flatten` pass, we cannot utilize that feature as the hierarchy may include UCs, which are described in a custom format not supported by Yosys.

Our tool is the first for SPFE that supports nesting modules inside other modules. Nested modules are implemented by instantiating other modules in the internal definition of a module. For merging the module hierarchy into a single circuit, we consider the hierarchy as a tree of module instantiations. Public modules that do not instantiate other modules are leaf nodes. Furthermore, we consider all private modules to be leaf nodes even though they might instantiate other modules, i.e., a private module's subtree of module instantiations is flattened. This prevents function designers from accidentally leaking implementation details of private sub-circuits. We summarize the algorithm for merging public and private modules in Algorithm 1.

Algorithm 1 Circuit merge algorithm

```
$nextWire \leftarrow 0$
$stack \leftarrow \{(\texttt{main}, 0, \emptyset, 0)\}$ // Begin with the main module, an empty translations set, and an offset of 0
while $stack \neq \emptyset$ do
    $(module, offset, translations, index) \leftarrow Pop(stack)$
    if $index = 0$ then
        $nextWire \leftarrow nextWire + MaxWire(module)$
    end if
    for $i = index, \ldots, module.cells.length$ do
        $cell = module.cells[i]$
        if $cell$ is a primitive gate then
            add $cell$ to the circuit, respecting $offset$ and $translations$
        else if $cell$ is a UC then
            add all UC gates to the circuit, respecting $offset$ and $translations$
        else
            set up submodule translations $subTranslations$
            $stack \leftarrow Push(stack, (module, offset, translations, i+1))$
            $stack \leftarrow Push(stack, (cell, nextWire, subTranslations, 0))$
            break
        end if
    end for
end while
```

Merging the hierarchy involves a depth-first search over the tree of instantiated modules. Starting with the `main` module, we iterate over all cells of the current module. Trivial logic cells (AND, XOR, and INV) are lowered to gates in the circuit. When encountering a new public module instantiation, we apply depth-first search by pushing our current progress to a stack and iterating deeper into the hierarchy. Private modules, on the other hand, are already compiled to UCs and can be added to the SPFE circuit without further descending into the hierarchy.

When adding a gate in a submodule to the resulting SPFE circuit, we need that gate's input and output wires to refer to the correct wire numbers. If module `foo` instantiates module `bar` and cell c_0 in `foo` refers to wire number 1 and cell c_1 in `bar` also refers to wire number 1, these wires are different. To disambiguate these wires, we add offsets to the respective wire numbers. With every instantiated module, we increase the offset by the maximum wire number in that module.

Another requirement for the correctness of the merge algorithm is the connection of submodule ports to their parent wires. This can either be achieved with copy gates or through wire renaming. FLUENT uses wire renaming to minimize the circuit size. Before descending into the submodule, we create a *translation* map which assigns every wire number associated with a port to the canonical wire number used in the final circuit.

Insertion of Conversion Gates. As the final circuit may comprise components evaluated through Yao's Garbled Circuits [20] and components evaluated via the SP-LUT [21] protocol, the compiler must insert gates to facilitate the necessary conversion between the two protocols. In particular, the inputs and outputs to public LUT gates are secret shared [30], whereas all other gate types operate on wire labels in Yao's garbled circuits protocol [20].

After all modules are merged, the tool performs a pass specific to the insertion of conversion gates. An integral part of this pass is the assignment of types to wires, i.e., every wire is either a Yao wire or an SP-LUT wire depending on whether the origin of this wire is a Yao gate (AND, XOR, INV, and UCs) or SP-LUT gate (LUTs). For every Yao gate, the compiler checks whether all input wires are Yao wires. Similarly, inputs to SP-LUT gates need to be SP-LUT wires. When this is not the case, additional conversion gates are added. This ensures that no unnecessary conversion gates are added.

F. SPFE Circuit Description Format

The output of our tool is an SPFE circuit description file with a `.spfe` extension. We intentionally designed the SPFE circuit format to be a superset of the UC circuit format of [18] to simplify parsing. Another consequence of this design is that every UC circuit is also a valid SPFE circuit, thus the same program used for evaluating SPFE functions can be reused for evaluating PFE functions. In addition to the circuit description file, a separate programming bit file with the extension `.spfe.prog` is generated, which includes the programming bits for the UC. We show an example SPFE circuit file in §B.

G. Evaluation of SPFE Circuits in ABY

ABY [19] is a framework for mixed-protocol two-party Secure Function Evaluation (SFE). It also implements the evaluation of Universal Circuits (UCs), thus enabling PFE [26], [27], [18]. We extended ABY to support SPFE circuits for UCs with multi-input gates. For this, we implemented a parser for SPFE circuit description files that builds internal ABY circuits from these descriptions. In particular, using multi-input gates in UCs required additional support in ABY.

The circuits generated by our FLUENT tool are not restricted to secure evaluation with ABY. As the circuit format is simple to parse, it can be easily integrated into other SFE frameworks such as MP-SPDZ [54], EZPC [55], and

MOTION [56]. We can apply Shannon's decomposition [57] to reduce a k-input LUT to a tree of $2^k - 1$ multiplexers in case the underlying SFE protocol does not natively support the evaluation of LUTs such as Yao's garbled circuit protocol [20]. The subsequent evaluation of multiplexers can be implemented using only AND and XOR gates as required by the underlying protocol [31]. The reduction works by creating a complete binary tree with $k+1$ layers with the nodes in the last layer representing the programming bit inputs. All other nodes are multiplexers that allow the selection of the correct programming bit for a specific input when combined.

In addition, we added support for hybrid Yao's garbled circuits [20] and SP-LUT [21] to the ABY framework [19]. Our SPFE compiler inserts the necessary conversion gates based on the SPFE circuit description file. As noted in [21], their LUT protocols can be freely combined with the GMW protocol [30] as both use XOR-based secret sharing. Therefore, the conversion routine between wire labels in Yao's GC scheme and wire shares in the SP-LUT protocol is identical to the conversions B2Y and Y2B between Yao's GC (Y) and the GMW protocol (B) [30], implemented already in ABY [19].

H. Toolchain

While our tool mainly consists of the `spfe` application, we provide additional tools for assistance in developing and testing semi-private functions.

spfe. The `spfe` application is the main interface to our tool. It converts a C/C++ or Verilog file passed on the command line to an SPFE circuit. The compilation process described in the preceding sections can be controlled with the `-l`, `-L`, and `-c` flags. Furthermore, debugging assistance is provided with the `-k` and `-v` flags.

UC. The `UC` executable is the UC compiler used in our pipeline. It is a modified version of the UC compiler implemented by [18]. The modifications made include improvements to the Bench [47] file parsing subroutine and a fix to ensure the order of output wires is preserved throughout the compilation.

convert. The `convert` tool performs direct circuit rewriting between different circuit formats. The supported input formats are Bristol [58] and SPFE (cf. §III-F), while Verilog and SPFE are supported as outputs. As circuits are translated without any modifications, the input circuit's wire numbers and gate types are preserved.

eval. By utilizing the `eval` tool, users can locally test their generated SPFE circuits using various input values (e.g., for debugging purposes). The `eval` tool operates by taking an SPFE circuit and input bits, and then printing the output in binary form. Compared to running an SFE protocol, circuit evaluation in clear text via `eval` is substantially faster and requires less memory. This tool is primarily employed for integration testing within our FLUENT tool.

IV. Evaluation

In this section, we describe our benchmarks of FLUENT.

Setup. We performed all benchmarks on two servers with Intel i9-7900X CPUs and 128 GB RAM[2] running Linux 6.0. Our modified version of the ABY framework [19] was used to measure the runtime and communication of evaluating the generated SPFE circuits. We used Yosys version 0.22 [24] in our compiler pipeline. The execution runtime results provided an average of over 10 executions.

Circuit	Public	Private	Overhead (×)
i32add	102	6 138	60.2
i32cmp	150	4 850	32.3
i32mul	1 948	133 354	68.5
i64add	235	14 853	63.2
i64cmp	322	11 134	34.6
i64mul	8 106	688 213	84.9
f32add	2 196	127 301	58.0
f32cmp	311	15 969	51.3
f32mul	2 791	190 774	68.4
sha1	29 633	4 016 545	135.5
sha256	62 172	8 046 181	129.4

TABLE I: Circuit sizes (number of AND gates) of various circuits implementing arithmetic and cryptographic operations in SFE (operations implemented as public modules) and PFE (operations implemented as private modules). The private circuit size is the number of AND gates of the smallest UC that implements the circuit. Overhead is the factor by which the private circuit is larger than the public circuit, i.e., the cost of hiding this circuit in the semi-private function.

A. SPFE Building Blocks

In the area of PFE, it is a well-known practice to benchmark arithmetic integer and floating points operations [59], [26], [27], [29], [18]. These circuits are frequently used as building blocks in many functions and, therefore, build the core of our benchmarks. Concretely, we benchmark each building block in both visibility classes, namely as public (SFE) and private (PFE) modules, and derive the additional cost to hide the building block in the function. In Tab. I, we provide the number of AND gates contained in different circuits when compiled as a public or private module. These circuits are addition, comparison, and multiplication operations for 32-bit and 64-bit unsigned integers and 32-bit IEEE floating-point numbers.

When looking at the largest circuit in Tab. I, sha256, we see that the number of AND gates of the private module is $\sim 129\times$ larger than the equivalent public module. Even the 32-bit integer addition as our smallest benchmark circuit has an overhead of factor $\sim 60\times$ when it is declared as private. Therefore, we can conclude that any effort spent on splitting the function into private and public modules, whenever possible, is always worthwhile as it will result in better evaluation performance.

B. Application: Car Insurance Tariff Calculation

To evaluate the practicality of our FLUENT tool, we benchmark the semi-private function example provided by [11] in our tool. This specific function represents a simple car insurance price calculation algorithm used for demonstration.

Car insurance tariff calculations are one of many important applications for SPFE as the insurance rate depends on sensitive private data such as accident history and car usage patterns. At the same time, it is in the interest of insurance

[2] For our benchmarks, less than 2 GB RAM would be sufficient.

Method	k_{pub}	k_{priv}	Circuit size	LAN Runtime [ms]		WAN Runtime [ms]		Communication [MB]	
				Setup	Online	Setup	Online	Setup	Online
SFE	Y	-	10 149	6.41	17.87	103.58	417.43	0.329	0.014
	4	-	-	3.09	112.20	93.69	4 732.93	0.008	0.030
	8	-	-	5.52	82.54	101.00	2 500.74	0.008	0.072
SPFE (Verilog)	Y	2	178 320	110.11	130.27	836.19	605.22	5.716	2.521
	Y	3	155 455	107.28	117.90	705.00	602.45	4.989	2.043
	Y	4	183 320	107.29	122.76	826.71	598.19	5.987	2.259
	Y	8	557 602	250.72	216.31	1 894.69	698.48	17.863	2.646
	Y	2’	178 320	104.46	113.37	820.68	603.80	5.715	2.525
	Y	3’	176 839	104.52	119.84	829.02	583.98	5.667	2.462
	Y	4’	203 281	109.70	117.93	837.56	706.16	6.513	2.791
	Y	8’	316 087	147.69	142.25	1 222.00	696.59	10.123	3.320
	4	2	-	101.59	192.20	798.06	3 286.61	5.342	2.539
	4	3	-	110.13	186.57	686.16	3 296.17	4.606	2.061
	4	4	-	111.53	190.76	822.02	3 275.43	5.613	2.277
	4	8	-	263.23	301.21	1 885.41	3 379.97	17.488	2.664
	8	2	-	115.22	173.87	785.14	1 933.31	5.342	2.604
	8	3	-	102.13	162.23	726.69	1 923.78	4.614	2.126
	8	4	-	105.17	171.54	834.02	1 900.63	5.613	2.343
	8	8	-	247.32	271.26	1 865.02	2 022.54	17.488	2.730
SPFE (C)	Y	2	308 922	140.11	167.12	1 215.30	758.84	9.894	4.204
	Y	3	251 180	124.83	142.55	999.02	706.25	8.046	3.117
	Y	4	280 197	131.85	135.37	1 078.68	718.35	8.975	3.211
	Y	8	809 386	340.90	295.86	2 561.35	878.21	25.909	3.805
SPFE [11]	-	-	358 170	185.60	207.18	1 326.06	888.04	11.470	5.129
improvement	-	-	2.01×	1.69×	1.59×	1.59×	1.47×	2.01×	2.03×
PFE	-	2	2 104 906	1 064.35	1 028.29	6 238.17	3 949.57	67.365	32.174
	-	3	1 694 241	891.66	908.10	5 097.26	3 148.36	54.224	24.732
	-	4	1 931 321	996.46	941.93	5 715.77	3 351.10	61.802	26.300
	-	8	4 105 627	1 889.46	1 502.48	11 833.01	3 642.35	131.389	24.296

TABLE II: Circuit sizes, runtime, and communication for private car insurance application. k_{pub} denotes the maximum input size of public LUT gates or Y when public modules are evaluated using Yao’s Garbled Circuits [20] protocol. Similarly, k_{priv} is the maximum number of inputs of private LUT gates. ’ in the k_{priv} column denotes usage of the VUC construction instead of LUC [18].

companies to hide the exact price calculation algorithm as they might otherwise lose a competitive advantage. However, many aspects of the insurance rate calculation are public knowledge and do not need to be hidden. For example, car insurance is often more expensive for people under a certain age. SPFE is an ideal protocol for evaluating this function.

To evaluate the example in FLUENT, minor changes to the original C source code were necessary. CBMC-GC [60], a compiler from ANSI C code into Boolean circuits initially built for SFE, was used in the SPFE framework of [11]. It allows the return of multiple values from a function by assigning them to variables with special names. As this feature is not compliant with the C standard, functions returning multiple values must be modified to return a struct containing these values. In addition to the car insurance calculator in C, we ported the complete code to Verilog for additional comparison. The car insurance calculator accepts 35 different input parameters of varying types and contains 15 modules (excluding `main`), 6 of which are private. Additionally, modules for fixed point arithmetic (multiplication, division, and exponentiation with an integer exponent) are included.

In the following benchmarks, we mostly investigate the LUC construction [18] (cf. §II) as this choice achieved the overall best performance while hiding the exact number of inputs for each gate in private modules. We also provide measurements for the VUC construction [18], which generates larger circuits for all numbers of LUT inputs of private modules k_{priv} tested, except for 8 LUT inputs.

To examine the performance gap between full PFE and SPFE, we compare the original circuit to a modified version where the entire module hierarchy is private. We also compile public modules with $k_{\text{pub}} \in \{4, 8\}$ input LUTs or $k_{\text{pub}} =$ Y for half-gates Yao [23], and private modules with $k_{\text{priv}} \in \{2, 3, 4, 8\}$ input LUTs. To analyze the performance impact of SPFE, we measure runtime and communication and compare them with the recent SPFE framework of [11] in Tab. II.

We notice dramatic performance improvements in runtime over PFE across the board when using SPFE. For $k_{\text{pub}} =$ Y and $k_{\text{priv}} = 3$, the number of AND gates in the PFE circuit is 1,694,241, while the SPFE circuit contains 155,455 AND gates, which is an improvement of $\sim 10.8\times$. The evaluation runtime and communication amount, which are proportional to the number of AND gates, tell a similar story. For LAN, the online runtime is reduced from 908 ms to 118 ms ($\sim 7.7\times$ improvement). For WAN, the online runtime is improved from 3,148 ms to 602 ms ($\sim 5.2\times$ improvement).

The communication in the setup phase has an even better improvement of $\sim$ 10.9× (54.224 MB for PFE, 5.0 MB for SPFE), which is closely tied to the reduction in circuit size. Evaluating the function using a mixed-protocol approach results in slightly improved setup communication ($\sim$ 5.0 MB for $k_{\text{pub}} =$ Y, $\sim$ 4.6 MB for $k_{\text{pub}} = 4$). However, due to the SP-LUT [21] protocol requiring a non-constant number of communication rounds, Yao outperforms the mixed-protocol approach in online runtime.

In addition, we observe that a LUT size of $k_{\text{priv}} = 3$ achieves the best overall results. For $k_{\text{priv}} > 4$, the gates added by the UC compiler to hide the circuit topology outweigh the efficiency gains resulting from the use of multi-input gates, which matches the observations made by [18].

Comparison to [11]. A final observation is that we can also compile more efficient circuits than the CBMC-GC-based SPFE framework [11]. Using our Verilog port of the function with $k_{\text{priv}} = 3$ and $k_{\text{pub}} = \text{Y}$, we deliver a $\sim 2\times$ circuit size improvement, which translates to a $\sim 1.59\times$ reduction in online runtime. The adapted version of the original C code yields a size improvement of $\sim 29.8\%$ and an online runtime improvement of $\sim 31.2\%$. These improvements stem from Liu et al.'s UC construction [29], which provides smaller circuits than Valiant's construction [8] used by [11], but also from the usage of multi-input LUTs.

C. Application: Convolutional Neural Network

We benchmark a C++ implementation of a simple Convolutional Neural Network (CNN) with a private convolution layer followed by a publicly fully connected layer. Although this small example is of limited practical use, it is a proof-of-concept of machine learning models with a mixture of public and private components. Compared to the fully private circuit, the semi-private circuit is $\sim 2.3\times$ smaller using $k_{\text{priv}} = 2$ and $k_{\text{pub}} = \text{Y}$. Online runtime is reduced by $\sim 2.4\times$ in a LAN setting.

V. Related Work

A. Private Function Evaluation (PFE)

In this section, we summarize PFE approaches beyond these based on the evaluation of a Universal Circuit (UC) with an SFE protocol that we already summarized in §II.

A natural way to solve PFE is Fully Homomorphic Encryption (FHE) [61], [62], where a party encrypts their private input x using their public key pk such that $\hat{x} = E_{pk}(x)$, gives $\hat{x}$ to the other party who then computes $\hat{y} = E_{pk}(f(\hat{x}))$, which the first party can decrypt with their private key sk to $y = D_{sk}(\hat{y})$. In contrast to other PFE approaches, the efficiency of FHE does not directly depend on the size of the computed function f, but on the size of the inputs and outputs [63]. However, the communication complexity is blown up to a polynomial degree in the length of the ciphertexts (the security parameter λ). Further, circuits are in general deep and hence require expensive bootstrapping to achieve circuit privacy [64], [65], [66].

Katz and Malka [67] proposed an approach for two-party PFE that is based on additively Homomorphic Encryption (HE) and has linear complexity in the size of the function. Their scheme was optimized and implemented by [68] showing that it improves over UC-based PFE already for circuits with a few thousand gates. However, HE-based PFE is not a very flexible solution as it cannot directly be combined with SFE frameworks and SFE-specific optimizations such as secure outsourcing [69] are not supported.

B. Semi-Private Function Evaluation (SPFE)

The work most relevant to our work is FairplaySPF [10]. FairplaySPF [10] extends the Fairplay SFE framework [70] to evaluate semi-private functions. It offers flexible control over function hiding, from SFE (no hiding) to PFE (complete hiding). They use privately programmable blocks (PBBs) within a public circuit topology. PBBs emulate functions from a set $\mathcal{F}$ based on hidden programming bits.

Users compose semi-private functions in the custom *Secure Programmable Block Description Language* (SPBDL). While its syntax is straightforward, it poses a steep learning curve for semi-private function programming. Unlike the clear parameters and return values in C/C++ functions that represent module connections in our tool, SPBDL relies on wire numbers, making it less intuitive.

A further drawback of the SPBDL format is its lack of modularity: If a user programs `foo.spbdl` and wants to reuse its functionality in a larger SPFE project, direct module referencing is not supported. This requires manual copying of SPBDL code into the larger function, which requires the user to recalculate wire numbers — a cumbersome process. In contrast, our tool allows to reuse `foo` throughout the code via simple function calls.

Furthermore, although UCs can be included as private modules, they must be external files and cannot reside in the SPBDL file itself. This makes it difficult to change the visibility of a certain subcomponent quickly. In our FLUENT tool, this only requires adding or removing the `spfe_private` pragma of the function in question. FairplaySPF [10] also presents an optimization to their construction, which reduces gate sizes for specific PBBs when parts of their input are known at circuit compile time. A similar optimization named `opt_expr` is included in the Yosys [24] synthesis suite. This optimization pass is included in our Yosys workflows (cf. §III-D).

Another SPFE framework uses the CBMC-GC compiler [60] for SFE, which translates ANSI C code into Boolean circuits. Public and private components are implemented in separate source files, and the overall circuit is assembled using a *Merger* tool. Like FairplaySPF [10], this tool requires the function designer to manually assign wire numbers to the input and output interfaces of the individual ANSI C files and keep track of them during the circuit design. This method is cumbersome and requires revising the entire circuit description provided to the merger tool if the interfaces of a single ANSI C file are changed or the visibility (public or private) changes. Consequently, using this framework is inconvenient.

In comparison to these two mentioned works, we provide the first SPFE framework that is both usable and allows the efficient evaluation of LUTs, thus improving the efficiency of SPFE.

C. Secure Function Evaluation (SFE)

SFE frameworks have seen significant research activity, particularly over the past decade. They have proven efficient in addressing privacy concerns across various real-world applications, including financial services [71], federated learning [72], [73], [74], [75], [76], [77], and privacy-preserving machine learning [78], [79], [80], [81], [82]. Compilers that translate high-level code into binary circuits offer an abstraction layer for SFE in the binary domain. Notable works in this area include Fairplay [70], FairplayMP [83], TASTY [84], VMCrypt [85], FastGC [86], Billion Gate malicious Yao [87], CBMC-GC [88], PCF [89], Obli-VM [90], HyCC [91], LLVM-MPC [92], SynCirc [93], and HyCaMi [94]. An in-depth overview and classification of SFE frameworks, including their specific details, can be found in [95]. The circuits generated by FLUENT are not restricted to evaluation in ABY [19]; they can be seamlessly integrated in various other SFE frameworks [96], [97], [90], [54], [56], [55], [98]. Another approach involves using existing hardware synthesis tools that take hardware description language (HDL) code, such as Verilog, as input. Examples include TinyGarble [99], TinyGMW [100], and SynCirc [93]. As showcased in [21] and FLUTE [36], this approach can be extended to LUT by repurposing LUT-based synthesis tools. We currently benchmark FLUENT with the LUT protocols of [21]. Using those of FLUTE [36] would improve performance even further.

In this rich landscape of SFE frameworks, GarbledCPU [101] is as an alternative method for hiding executed functions. In GarbledCPU, a CPU core circuit is garbled and executed with private inputs corresponding to the CPU core's instructions, often implementing established instruction sets such as MIPS [101], [102], [12] and ARM [103], [12]. Subsequently, the binary representation of the secure function is loaded onto the circuit, allowing users to program in a language of their choice. However, GarbledCPU cannot be directly compared with FLUENT because it uses a different notion of SPFE. In FLUENT, we separate circuits into public and private components, hiding only the private sub-circuits. In contrast, achieving such a clear separation is not feasible in GarbledCPU. Here, the term "semi-private" refers to certain instructions being removed from the instruction set. However, excluding operations like multiplications from an instruction set containing addition and shift instructions does not inherently reveal details about executed functions. Multiplication operations can be restructured using additions and shifts. In contrast, FLUENT does not simulate a CPU core, but it provides precise control over the privacy of distinct function components, offering a level of granularity unattainable in GarbledCPU. This flexibility suggests the potential for integration into a CPU-based SPFE framework by migrating public function components to custom instructions within established instruction sets, like RISC-V. To our knowledge, no such framework currently exists. Consequently, a direct one-to-one comparison lacks meaningful context.

Moreover, GarbledCPU relies on an MPC backend that supports sequential circuits, a feature shared with frameworks like TinyGarble [99], [104] and FPGA-based circuit evaluators used in both GarbledCPU [102] and Garbled EDA [12]. However, the majority of MPC frameworks primarily support only combinational circuits, including ABY [19], ABY 2.0 [34], MP-SPDZ [54], and MOTION [56]. FLUENT is fully compatible with these frameworks.

VI. Conclusion

In this paper, we provide a tool for programming semi-private functions in the C/C++ programming language. With pragmas, programmers can easily control a component's visibility as public (SFE) or private (PFE). With that, we improve the usability over earlier frameworks by seamlessly combining the structural description of the function with the functionality of the components in a single source file. Furthermore, we demonstrate through benchmarking that the use of SPFE results in significant performance improvements over PFE. In future work, we will engineer a new compiler/toolchain that generates LUT circuits that are fine-tuned for LUT-based protocols, as the ones utilized in this work were generated using standard hardware development tools instead of custom solutions. It would also be interesting to investigate the benefits of a hybrid approach that combines LUTs and standard arithmetic gates, which can locally compute additions and compute multiplications interactively, for SFE. In terms of applications, it would be intriguing to investigate the impact of leveraging our FLUENT tool with recent works on privacy-preserving machine learning, federated learning, and logic locking.

Acknowledgments

This project received funding from the ERC under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the DFG within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230.

References

[1] D. Günther, J. Schmidt, T. Schneider, and H. Yalame, "FLUENT: A Tool for Efficient Mixed-Protocol Semi-Private Function Evaluation," in *ACSAC*, 2024.

[2] J. Pecholt and S. Wessel, "Cocotpm: Trusted platform modules for virtual machines in confidential computing environments," in *ACSAC*, 2022.

[3] K. Shamsi, D. Z. Pan, and Y. Jin, "On the impossibility of approximation-resilient circuit locking," in *HOST*, 2019.

[4] J. A. Roy, F. Koushanfar, and I. L. Markov, "Ending piracy of integrated circuits," *Computer*, 2010.

[5] E. Masserova, D. Garg, K. Mai, L. Pileggi, V. Goyal, and B. Parno, "Logic Locking-Connecting Theory and Practice," 2022.

[6] P. Beerel, M. Georgiou, B. Hamlin, A. J. Malozemoff, and P. Nuzzo, "Towards a Formal Treatment of Logic Locking," *TCHES*, 2022.

[7] V. Kolesnikov and T. Schneider, "A Practical Universal Circuit Construction and Secure Evaluation of Private Functions," in *FC*, 2008.

[8] L. G. Valiant, "Universal circuits (Preliminary Report)," in *STOC*, 1976.

[9] I. Wegener, "The Complexity of Boolean Functions," in *Complexity Theory: Exploring the Limits of Efficient Algorithms*, 2005.

[10] A. Paus, A.-R. Sadeghi, and T. Schneider, "Practical Secure Evaluation of Semi-private Functions," in *ACNS*, 2009.

[11] D. Günther, Á. Kiss, L. Scheidel, and T. Schneider, "Poster: Framework for Semi-Private Function Evaluation with Application to Secure Insurance Rate Calculation," in *CCS*, 2019.

[12] M. Hashemi, S. Roy, F. Ganji, and D. Forte, "Garbled EDA: Privacy Preserving Electronic Design Automation," in *ICCAD*, 2022.

[13] K. Frikken, M. Atallah, and C. Zhang, "Privacy-Preserving Credit Checking," in *Electronic Commerce (EC)*, 2005.

[14] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider, "Secure Evaluation of Private Linear Branching Programs with Medical Applications," in *ESORICS*, 2009.

[15] R. Ostrovsky and W. E. Skeith, "Private Searching on Streaming Data," in *JoC*, 2007.

[16] M. Hashemi, S. Roy, D. Forte, and F. Ganji, "HWGN^2: Side-Channel Protected NNs Through Secure and Private Function Evaluation," in *Security, Privacy, and Applied Cryptography Engineering (SPACE)*, 2022.

[17] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel, "Privacy-Preserving Remote Diagnostics," in *CCS*, 2007.

[18] Y. Disser, D. Günther, T. Schneider, M. Stillger, A. Wigandt, and H. Yalame, "Breaking the Size Barrier: Universal Circuits meet Lookup Tables," in *ASIACRYPT*, 2023.

[19] D. Demmler, T. Schneider, and M. Zohner, "ABY–A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *NDSS*, 2015.

[20] A. C. Yao, "How to Generate and Exchange Secrets," in *FOCS*, 1986.

[21] G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, "Pushing the Communication Barrier in Secure Computation using Lookup Tables," in *NDSS*, 2017.

[22] "XLS: Accelerated HW Synthesis," https://google.github.io/xls, 2020.

[23] S. Zahur, M. Rosulek, and D. Evans, "Two Halves make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates," in *EUROCRYPT*, 2015.

[24] C. Wolf, "Yosys Open SYnthesis Suite," https://yosyshq.net/yosys/, 2016.

[25] A. Sadeghi and T. Schneider, "Generalized Universal Circuits for Secure Evaluation of Private Functions with Application to Data Classification," in *ICISC*. Springer, 2008.

[26] D. Günther, Á. Kiss, and T. Schneider, "More Efficient Universal Circuit Constructions," in *ASIACRYPT*, 2017.

[27] M. Y. Alhassan, D. Günther, Á. Kiss, and T. Schneider, "Efficient and Scalable Universal Circuits," *JoC*, 2020.

[28] S. Zhao, Y. Yu, J. Zhang, and H. Liu, "Valiant's Universal Circuits Revisited: An Overall Improvement and a Lower Bound," in *ASIACRYPT*, 2019.

[29] H. Liu, Y. Yu, S. Zhao, J. Zhang, W. Liu, and Z. Hu, "Pushing the Limits of Valiant's Universal Circuits: Simpler, Tighter and More Compact," in *CRYPTO*, 2021.

[30] O. Goldreich, S. Micali, and A. Wigderson, "How to Play Any Mental Game or a Completeness Theorem for Protocols with Honest Majority," in *STOC*, 1987.

[31] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[32] T. Schneider and M. Zohner, "GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits," in *FC*, 2013.

[33] H. Yalame, H. Farzam, and S. Bayat-Sarmadi, "Secure Two-Party Computation Using an Efficient Garbled Circuit by Reducing Data Transfer," in *Applications and Techniques in Information Security*, 2017.

[34] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation," in *USENIX Security*, 2021.

[35] A. Brüggemann, O. Schick, T. Schneider, A. Suresh, and H. Yalame, "Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater," in *IEEE S&P*, 2024.

[36] A. Brüggemann, R. Hundt, T. Schneider, A. Suresh, and H. Yalame, "FLUTE: fast and secure lookup table evaluations," in *IEEE S&P*, 2023.

[37] C. Harth-Kitzerow, A. Suresh, Y. Wang, H. Yalame, G. Carle, and M. Annavaram, "High-Throughput Secure Multiparty Computation with an Honest Majority in Various Network Settings," in *PETS*, 2025.

[38] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *STOC*, 1990.

[39] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient Garbling from a Fixed-Key Blockcipher," in *IEEE S&P*, 2013.

[40] M. Rosulek and L. Roy, "Three Halves Make a Whole? Beating the Half Gates Lower Bound for Garbled Circuits," in *CRYPTO*, 2021.

[41] Intel Corporation, "Intel® High Level Synthesis Compiler," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html, 2018.

[42] Microchip Technology Inc., "SmartHLS Compiler," https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler, 2021.

[43] "Advanced Micro Devices, Xilinx Vivado," https://www.xilinx.com/products/design-tools/vivado.html, 2022.

[44] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification," https://people.eecs.berkeley.edu/∼alanmi/abc/, 2013.

[45] Cho, Sungmin and Chatterjee, Satrajit and Mishchenko, Alan and Brayton, Robert, "Efficient FPGA Mapping using Priority Cuts," in *Proc. FPGA*, 2007.

[46] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," Tech. Rep. RFC 8259, 2017.

[47] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," in *International Symposium on Circuits and Systems (ISCAS)*, 1989.

[48] I. Page, "Closing the Gap between Hardware and Software: Hardware-Software Cosynthesis at Oxford," in *IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*, 1996.

[49] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Symposium on Code Generation and Optimization (CGO)*, 2004.

[50] "IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language," in *IEEE Standard*, 2005.

[51] Zig Software Foundation, "Zig Programming Language," https://ziglang.org, 2016.

[52] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS," 2003, www.openssl.org.

[53] A. Mishchenko, "Berkeley Logic Interchange Format (BLIF)," Tech. Rep., 1992.

[54] M. Keller, "MP-SPDZ: A Versatile Framework for Multi-Party Computation," in *CCS*, 2020.

[55] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning," in *EuroS&P*, 2019.

[56] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "MOTION - A Framework for Mixed-Protocol Multi-Party Computation," *TOPS*, 2022.

[57] G. Boole, "An Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities." Walton and Maberly, 1854.

[58] S. Tillich and N. Smart, "(Bristol Format) Circuits of Basic Functions Suitable For MPC and FHE," https://homes.esat.kuleuven.be/∼nsmart/MPC/old-circuits.html, 2011.

[59] Á. Kiss and T. Schneider, "Valiant's Universal Circuit is Practical," in *EUROCRYPT*, 2016.

[60] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations," in *Compiler Construction (CC)*, 2014.

[61] C. Gentry, "Fully Homomorphic Encryption using Ideal Lattices," in *STOC*, 2009.

[62] S. Halevi and V. Shoup, "Faster Homomorphic Linear Transformations in HElib," in *CRYPTO*, 2018.

[63] Z. Brakerski, N. Döttling, S. Garg, and G. Malavolta, "Leveraging Linear Decryption: Rate-1 Fully-Homomorphic Encryption and Time-Lock Puzzles," in *TCC*, 2019.

[64] C. Gentry and S. Halevi, "Compressible FHE with Applications to PIR," in *TCC*, 2019.

[65] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE," in *ASIACRYPT*, 2017.

[66] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs," in *TCHES*, 2021.

[67] J. Katz and L. Malka, "Constant-Round Private Function Evaluation with Linear Complexity," in *ASIACRYPT*, 2011.

[68] M. Holz, Á. Kiss, D. Rathee, and T. Schneider, "Linear-Complexity Private Function Evaluation is Practical," in *ESORICS*, 2020.

[69] S. Kamara and M. Raykova, "Secure Outsourced Computation in a Multi-tenant Cloud," in *IBM Workshop on Cryptography and Security in Clouds*, 2011.

[70] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella *et al.*, "Fairplay-Secure Two-Party Computation System." in *USENIX Security*, 2004.

[71] S. Atapoor, N. P. Smart, and Y. T. Alaoui, "Private Liquidity Matching using MPC," in *CT-RSA*, 2022.

[72] H. Fereidooni, S. Marchal, M. Miettinen, A. Mirhoseini, H. Möllering, T. D. Nguyen, P. Rieger, A.-R. Sadeghi, T. Schneider, H. Yalame *et al.*, "SAFELearn: Secure Aggregation for Private Federated Learning," in *DLS@IEEE S&P*, 2021.

[73] T. D. Nguyen, P. Rieger, R. De Viti, H. Chen, B. B. Brandenburg, H. Yalame, H. Möllering, H. Fereidooni, S. Marchal, M. Miettinen *et al.*, "FLAME: Taming Backdoors in Federated Learning," in *USENIX Security*, 2022.

[74] F. Marx, T. Schneider, A. Suresh, T. Wehrle, C. Weinert, and H. Yalame, "WW-FL: Secure and Private Large-Scale Federated Learning," *https://arxiv.org/abs/2302.09904*, 2023.

[75] T. Gehlhar, F. Marx, T. Schneider, A. Suresh, T. Wehrle, and H. Yalame, "SafeFL: MPC-Friendly Framework for Private and Robust Federated Learning," in *DLS@IEEE S&P*, 2023.

[76] M. Rathee, C. Shen, S. Wagh, and R. A. Popa, "ELSA: Secure Aggregation for Federated Learning with Malicious Actors," in *IEEE S&P*, 2023.

[77] Y. Ben-Itzhak, H. Möllering, B. Pinkas, T. Schneider, A. Suresh, O. Tkachenko, S. Vargaftik, C. Weinert, H. Yalame, and A. Yanai, "Scionfl: Efficient and Robust Secure Quantized Aggregation," in *IEEE SaTML*, 2024.

[78] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-Party Secure Inference," in *CCS*, 2020.

[79] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, “Mp2ml: A mixed-protocol machine learning framework for private inference,” in *ARES*, 2020.

[80] H. Keller, H. Möllering, T. Schneider, and H. Yalame, “Balancing Auality and Efficiency in Private Clustering with Affinity Propagation,” in *SECRYPT*, 2021.

[81] A. Hegde, H. Möllering, T. Schneider, and H. Yalame, “SoK: Efficient Privacy-Preserving Clustering,” in *PETS*, 2021.

[82] V. Duddu, A. Das, N. Khayata, H. Yalame, T. Schneider, and N. Asokan, “Attesting Distributional Properties of Training Data for Machine Learning,” in *ESORICS*, 2024.

[83] A. Ben-David, N. Nisan, and B. Pinkas, “FairplayMP: A System for Secure Multi-Party Computation,” in *CCS*, 2008.

[84] W. Henecka, S. Kögl, A. R. Sadeghi, T. Schneider, and I. Wehrenberg, “TASTY: Tool for Automating Secure Two-PartY Computations,” in *CCS*, 2010.

[85] L. Malka, “VMCrypt: Modular Software Architecture for Scalable Secure Computation,” in *CCS*, 2011.

[86] Y. Huang, D. Evans, J. Katz, and L. Malka, “Faster Secure Two-Party Computation Using Garbled Circuits.” in *USENIX Security*, 2011.

[87] B. Kreuter, A. Shelat, and C.-H. Shen, “Billion-Gate Secure Computation with Malicious Adversaries,” in *USENIX Security*, 2012.

[88] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, “Secure two-party computations in ANSI C,” in *CCS*, 2012.

[89] B. Kreuter, A. Shelat, B. Mood, and K. R. B. Butler, “PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation,” in *USENIX*, 2013.

[90] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, “ObliVM: A Programming Framework for Secure Computation,” in *IEEE S&P*, 2015.

[91] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, “HyCC: Compilation of Hybrid Protocols for Practical Secure Computation,” in *CCS*, 2018.

[92] T. Heldmann, T. Schneider, O. Tkachenko, C. Weinert, and H. Yalame, “LLVM-Based Circuit Compilation for Practical Secure Computation,” in *ACNS*, 2021.

[93] A. Patra, T. Schneider, A. Suresh, and H. Yalame, “SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation,” in *IEEE HOST*, 2021.

[94] H. Mantel, J. Schmidt, T. Schneider, M. Stillger, T. Weißmantel, and H. Yalame, “HyCaMi: High-Level Synthesis for Cache Side-Channel Mitigation,” in *DAC*, 2024.

[95] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, “SoK: General Purpose Compilers for Secure Multi-Party Computation,” in *IEEE S&P*, 2019.

[96] X. Wang, A. J. Malozemoff, and J. Katz, “EMP-toolkit: Efficient MultiParty computation toolkit,” https://github.com/emp-toolkit, 2016.

[97] S. Zahur and D. Evans, “Obliv-C: A Language for Extensible Data-Oblivious Computation,” 2015.

[98] J.-P. Münch, T. Schneider, and H. Yalame, “VASA: Vector AES Instructions for Security Applications,” in *ACSAC*, 2021.

[99] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, “TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits,” in *IEEE S&P*, 2015.

[100] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni, “Automated Synthesis of Optimized Circuits for Secure Computation,” in *CCS*, 2015.

[101] X. Wang, S. D. Gordon, A. McIntosh, and J. Katz, “Secure Computation of MIPS Machine Code,” in *ESORICS*, 2016.

[102] E. M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar, “GarbledCPU: A MIPS Processor for Secure Computation in Hardware,” in *DAC*, 2016.

[103] E. M. Songhori, M. S. Riazi, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, “ARM2GC: Succinct Garbled Processor for Secure Computation,” in *DAC*, 2019.

[104] S. Hussain, B. Li, F. Koushanfar, and R. Cammarota, “TinyGarble2: Smart, Efficient, and Scalable Yao’s Garble Circuit,” in *Workshop on Privacy-Preserving Machine Learning in Practice @ CCS*, 2020.

Appendix A
Source Code Example

Listing 5: Example C++ HLS input code.

```
#pragma spfe_private
int age_premium(int age) {
  if (age < 21) {
    return 100;
  } else {
    return 0;
  }
}
int base_price = 1000;

int main(int age) {
  return base_price + age_premium(age);
}
```

List. 5 shows C++ code for an age-dependent price calculation and List. 6 the converted Verilog code after using XLS [22]. This example illustrates the value of incorporating HLS into our pipeline. The input C++ code describes a simple semi-private function containing a private module. XLS [22] converts the function call to `age_premium` to a module instantiation of the respective Verilog module. Familiar language is crucial for a good user experience.

Listing 6: Output Verilog generated by XLS[22].

```
 (* private *)
module _Z11age_premiumi(
  input wire [31:0] age,
  output wire [31:0] out
);
  assign out = {32{$signed(age) < $signed
      (32'h0000_0015)}} & 32'h0000_0064;
endmodule

module main(
  input wire [31:0] age,
  output wire [31:0] out
);
  wire [31:0] instantiation_output_83;
  wire [28:0] add_78;
  assign add_78 = instantiation_output_83
      [31:3] + 29'h0000_007d;
  _Z11age_premiumi invoke_75 (
    .age(age),
    .out(instantiation_output_83)
  );
  assign out = {add_78,
      instantiation_output_83[2:0]};
endmodule
```

Appendix B
SPFE circuit file example in FLUENT

We briefly show a manually crafted example of an SPFE circuit file in List. 7 to demonstrate the syntax of our circuit format in FLUENT. The SPFE circuit format is line-based, i.e. each line describes inputs, outputs, or a circuit gate. Lines are made up of fields separated by whitespace. The value of the first field determines the object the line represents.

When the first character is `C`, the remaining fields represent a list of input wire numbers. Similarly, lines starting with `O` represent circuit outputs.

Lines describing gates first consist of their type, then a list of inputs, and finally a list of outputs. Boolean gates evaluated in Yao’s Garbled Circuits [20], [23] protocol are represented with `A` (AND gates), `E` (XOR gates), and `I` (NOT gates).

When the mixed-protocol approach is enabled, the gate types `S`, `B`, and `L` are used as well. An `S` gate converts a wire label in Yao's Garbled Circuit protocol [20], [23] to a Boolean share [21]; the `B` gates performs the inverse operation, i.e., the conversion from Boolean share to Yao's Garbled Circuit. `L` gates are k_{pub}-input LUT gates. The corresponding table for these gates is provided in the last field.

Universal circuits consist of `X`, `Y`, and `U` gates. All three gate types require programming known only to the party owning the function. An `X` gate has two inputs and two outputs and forwards its inputs to its outputs either in the same order (when programmed with 0) or swapped (when programmed with 1). `Y` gates have two inputs and one output and forward either the left input (when programmed with 1) or the right input (when programmed with 0) to their output. `U` gates are k_{priv}-input LUTs which are programmed with 2^k programming bits.

Listing 7: Example of an SPFE circuit file

```
C 0 1 2 3
A 0 1 4
I 4 5
U 2 3 6
E 5 6 7
S 0 8
S 7 9
L 8 9 10 0001
B 10 11
O 11
```