SoK: General Purpose Compilers for Secure Multi-Party Computation

Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic
University of Pennsylvania
{ mhast, fbrett, dgnoble, stevez } @cis.upenn.edu

***Abstract*—Secure multi-party computation (MPC) allows a group of mutually distrustful parties to compute a joint function on their inputs without revealing any information beyond the result of the computation. This type of computation is extremely powerful and has wide-ranging applications in academia, industry, and government. Protocols for secure computation have existed for decades, but only recently have general-purpose compilers for executing MPC on arbitrary functions been developed. These projects rapidly improved the state of the art, and began to make MPC accessible to non-expert users. However, the field is changing so rapidly that it is difficult even for experts to keep track of the varied capabilities of modern frameworks.**

In this work, we survey general-purpose compilers for secure multi-party computation. These tools provide high-level abstractions to describe arbitrary functions and execute secure computation protocols. We consider eleven systems: EMP-toolkit, Obliv-C, ObliVM, TinyGarble, SCALE-MAMBA (formerly SPDZ), Wysteria, Sharemind, PICCO, ABY, Frigate and CBMC-GC. We evaluate these systems on a range of criteria, including language expressibility, capabilities of the cryptographic back-end, and accessibility to developers. We advocate for improved documentation of MPC frameworks, standardization within the community, and make recommendations for future directions in compiler development. Installing and running these systems can be challenging, and for each system, we also provide a complete virtual environment (Docker container) with all the necessary dependencies to run the compiler and our example programs.

I. Introduction

Secure multi-party computation (MPC) provides a mechanism by which a group of data-owners can compute joint functions of their private data, where the execution of the protocol reveals nothing more about the underlying data than what is revealed by the output alone. MPC can be viewed as a cryptographic method for providing the functionality of a trusted party—who would accept private inputs, compute a function and return the result to the stakeholders—without the need for mutual trust.

Thanks to these strong security guarantees, MPC has broad potential for practical applications, ranging from general computations of secure statistical analysis [23], [24], [26], [57], [58], [59], [60], [100], to more domain-specific uses like financial oversight [2], [22], [27], [63], biomedical computations [38], [34], [80], [88], [86], [89], [136] and satellite collision detection [78], [79], [90].

Despite the demand for MPC technology, practical adoption has been limited, partly due to the *efficiency* of the underlying protocols. General-purpose MPC protocols, capable of securely computing *any* function, have been known to the cryptographic community for 30 years [33], [73], [131], [132]. Until recently such protocols were mainly of theoretical interest, and were considered too inefficient (from the standpoint of computation and communication complexity) to be useful in practice.

To address efficiency concerns, cryptographers have developed highly-optimized, special-purpose MPC protocols for a variety of use-cases. Unfortunately, this mode of operation does not foster widespread deployment or adoption of MPC in the real world. Even if these custom-tailored MPC protocols are theoretically *efficient* enough for practical use, designing, analyzing and implementing a custom-tailored protocol from the ground up for each application is not a scalable solution.

General-purpose MPC *compilers*, could drastically reduce the burden of designing multiple custom protocols and could allow non-experts to quickly prototype and deploy secure computations. Using compilers, the engineering effort devoted to making general-purpose MPC protocols practical and secure can be amortized across all of the uses of such a system.

Many significant challenges arise when designing and building an MPC compiler. In general, implementing any type of multi-round, distributed protocol robustly and efficiently is a major engineering challenge, but the MPC compilers have additional requirements that make them especially challenging to build correctly. For efficiency, both the compiler and the cryptographic back-end need to be highly optimized. For usability, the front-end compiler needs to be expressive, flexible, and intuitive for non-experts, and should abstract away many of the complexities of the underlying MPC protocol, including circuit-level optimizations (e.g. implementing floating-point operations as a Boolean circuit) and back-end protocol choice (e.g. selecting an optimal protocol for a particular computation). With today's compilers, optimizing performance often still requires a fair degree

of knowledge and effort on the part of the user.

Fairplay [103] was the first publicly available MPC compiler. It translated code written in a high-level Secure Function Definition Language (SFDL) into a garbled circuit representation, which could then be (securely) evaluated by two parties. Fairplay was subsequently extended to allow for true multi-party computation in FairplayMP [15], using a modified version of the BMR protocol [10]. It was followed shortly by VIFF [68], [50] and SEPIA [29], which used the same basic architecture: they took programs written in fairly high-level languages, converted them to a circuit format, and executed the circuit using a secure computation protocol.

These early compilers showed that general-purpose MPC was achievable, and, although their performance rendered them unsuitable for most real-world applications, they launched what is now a very active field of research in MPC compiler design and implementation.

Thanks to these efforts, dramatic improvements in secure computation algorithms coupled with a steady increase in hardware performance have made MPC a viable solution to a large class of real-world problems. Modern MPC protocol implementations are fast enough to securely evaluate complex functions on moderately-large data sets, such as the numerous implementations of secure regression analyses with tens to hundreds of thousands of observations, and tens to hundreds of variables [24], [41], [67], [91], [108].

The rush of activity in this field can be difficult to navigate: dozens of new compilers and supporting frameworks encompass a wide variety of architectures and features which influence their efficiency, usability and suitability for different tasks. The goal of this Systematization of Knowledge paper is to provide a guide to the powerful new breed of MPC compilers, and is primarily aimed at four distinct types of readers.

1) Developers looking to choose a compiler with which to implement a specific secure computation
2) Theoretical cryptographers looking to understand state-of-the-art in practical, secure computation
3) Compiler designers looking to understand the limitations of existing technology and identify new research directions
4) Managers and policy-makers looking to understand whether existing technology is mature enough for their needs

This paper briefly reviews necessary technical background about secure multi-party computation, then surveys the state-of-the-art MPC frameworks, evaluating them based on their general architecture, underlying technology, threat models, and expressiveness. Our comparison focuses on usability features rather than performance metrics, and we report on our experience with implementing three small test programs in each case. Casual readers may wish to skim Section VI, which discusses each framework in greater depth, and focus on the final discussion section, where we advocate for improved documentation and standardization and suggest future directions in compiler research.

Many of these frameworks are themselves research projects or works-in-progress: they have non-trivial build dependencies and complicated work flows. Indeed, implementing our simple example programs in each system required significant engineering effort: we estimate over 750 person-hours. To allow others to experiment more easily with these systems, we have created an on-line Github repository[1] with two artifacts: (1) a set of Docker containers, each of which provides a development environment configured with the required software infrastructure for each MPC framework, along with executable examples of our test cases, and (2) a wiki page that collects much of the evaluation presented here with additional documentation about each framework.

a) Related Work: Archer et al.'s survey [6] of secure computation tools across several paradigms, including garbled circuit schemes, defines a maturity taxonomy that aims to describe the practical readiness of several schemes. Shan et al.'s survey [121] outlines different threat models and computation techniques for securely outsourcing many specific types of computation. The authors of Frigate [105] include a short survey of existing MPC frameworks, which focuses on correctness and covers a slightly older body of work. The SSC protocol comparison tool [109], [110], [111] allows users to find published protocols matching certain security or privacy criteria, but this tool classifies *theoretical* protocols rather than implementations and does not include protocols developed in the past few years. The awesome-mpc repository [119] provides an up-to-date list of compilers, back-ends and special-purpose protocols, with a short description of each. To the best of our knowledge, these previous works did not actually install and experiment with each of the systems they surveyed, but drew their conclusions based on the descriptions of the systems in their published papers and documentation. Unfortunately, we have found that the features, functionality and syntax of the actual implementations do not always match those found in the documentation.

II. Cryptographic Building Blocks

In this section, we describe important cryptographic primitives common to many MPC protocols.

A. Secret Sharing

Cryptographic secret sharing protocols [120], [21] allow a dealer to break a secret value into *shares* and distribute these shares to group of recipients with the

[1] https://github.com/MPC-SoK/frameworks

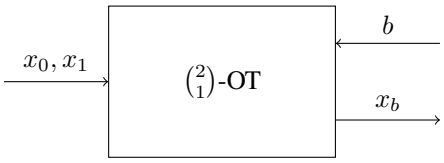

Fig. 1: Oblivious Transfer

property that any unqualified set of recipients learns nothing about the secret, while any qualified set of recipients can reconstruct the secret from their shares. In practice, most secret-sharing protocols are *threshold* protocols, where any collection of fewer than t shares reveals nothing about the secret – and any subset of size at least t can reconstruct the secret. Many general secret sharing schemes exist [18], [28], [20], [125], [45], as well as constructions of secret-sharing schemes for general (non-threshold) access structures [69], [81]. See Beimel [12] for a survey of secret-sharing protocols.

In practice, most MPC protocols rely on a linear secret sharing scheme: either simple, additive secret sharing or Shamir sharing. Both satisfy linearity: the sum of two secret shares is equal to the share of the sum.

B. Oblivious Transfer (OT)

Oblivious Transfer (OT) [114], [130], [62] is a cryptographic protocol that allows a party to choose k of n secrets from another party, without revealing which secret they have chosen. Figure 1 shows $\binom{2}{1}$-OT, where one secret is chosen from two options.

From a theoretical standpoint, MPC protocols can be built from OT alone [95], [84], but the key feature that makes OT suitable for building *efficient* MPC protocols is that OT is equivalent to a seemingly weaker functionality called *Random OT* (ROT) [46], where the choice bit b is not provided as an input, but instead is randomly generated by the protocol itself. The output of a ROT protocol is two correlated pairs of bits (x_0, x_1) and (b, x_b), where x_0, x_1, b are uniformly distributed in $\{0, 1\}$. Given a random OT correlation (the pairs (x_0, x_1) and (b, x_b)) Alice and Bob can compute the OT functionality using only three bits of communication.

Since ROT implies OT, parties can compute all the necessary ROTs needed for a protocol in advance, in an input-independent pre-processing ("offline") phase. Then, in the "online" phase, they consume these pre-generated OTs and execute the protocol with minimal communication cost and no computationally expensive public-key operations. This offline-online separation makes the online phase of such a protocol extremely efficient, but there is still the problem of making the pre-processing phase efficient. There are two fundamentally different approaches to handling the offline phase, either through a *trusted dealer* or a cryptographic batched correlation-generation protocol.

In the trusted dealer model, a semi-trusted dealer simply distributes correlated randomness to all the parties. The trusted dealer has no inputs, and never handles any secret information, thus the dealer need only be trusted to generate and distribute random values to the appropriate parties. In the presence of a trusted dealer, the offline phase of many MPC protocols can be extremely efficient.

Even without a trusted dealer, OTs can be generated efficiently through *OT extension*, where a small number of "base" or "seed" OTs are converted into a massive number of ROTs [83] through the use of efficient symmetric-key primitives (e.g. AES). Since its introduction, there have been many variants of OT-extension [7], [84], [77], [107], [98], [93] and OT-extension has become an essential feature in almost all MPC implementations.

III. Secure Multi-party Computation

Secure multi-party computation protocols (MPC) allow a group of mutually distrustful stakeholders to securely compute *any* function of their joint inputs in such a way that the execution of the protocol provably reveals nothing beyond the result of the computation. Security is often defined using a *simulation paradigm*, where a protocol is said to be secure if there exists an efficient (polynomial-time) simulator that takes as input the output of the computation and produces a protocol transcript (the "views" of each participant) such that any view (or some subsets of views) is indistinguishable from the transcript created by a real execution of the protocol.

This ensures that each participant (or certain colluding subsets of participants) learn nothing from executing the protocol that they could not have learned from the output alone. In this way, MPC cryptographically emulates a trusted party who accepts each participant's private input, computes the desired function and returns the result.

Early MPC protocols used a circuit model for secure computation, first representing the target function as either a Boolean or arithmetic circuit (over some finite field), then securely evaluating the circuit gate-by-gate. Many of the compilers we analyzed still use this circuit model of computation. In the remainder of this section, we sketch key design characteristics of the major protocol families that underpin modern MPC systems.

A. Garbled Circuits

Circuit garbling is a method for secure two-party computation, originally introduced by Yao [131], [132]. In this framework, there are two participants, a *garbler* and an *evaluator*. The participants begin by expressing the desired function as a *Boolean circuit*. The garbler then proceeds to garble the circuit gate-by-gate using a

standard symmetric-key cryptosystem (usually AES), as follows. For each wire in the circuit, the garbler selects two uniformly random and independent "wire tokens." The garbler then expresses each gate in the circuit as a truth table by encrypting each output wire token with the two input wire tokens that generate it (for Boolean circuits with fan-in two, each truth table will have four rows). The garbler permutes the rows of the truth-table, and sends the entire collection of "garbled" gates to the evaluator. The garbling procedure is designed so that learning one wire token for each input wire of a gate allows you to decrypt exactly one row of its garbled truth-table, revealing a single wire token belonging to the output-wire of that gate. Thus an evaluator that learns a single wire token for each input wire of the circuit can iteratively produce the wire token for each wire in the circuit, and completely evaluate the circuit.

To provide its secret inputs, the garbler sends the correct wire tokens directly to the evaluator. For each bit of the evaluator's inputs, the garbler and evaluator engage in an oblivious transfer where the evaluator secretly selects one of two wire tokens offered by the garbler. Once the evaluator has a wire token for each input bit, it can evaluate the circuit (performing symmetric-key decryptions for every gate) and learn the result. In the semi-honest model, the evaluator forwards the result of the computation to the garbler. For a formal description of the garbling procedure, see Bellare *et al.* [14].

The initial garbled circuit protocol provided security against semi-honest adversaries [99], but there exist many different improvements and implementations that provide security against fully malicious adversaries.

Two key performance improvements are the "free XOR trick" [97], which evaluates XOR gates in a single round without any cryptographic operations required by the garbled tables; and Half-Gates [134], which reduce the number of ciphertexts required to garble AND gates. The addition of the AES-NI instruction set made computing AES encryptions on modern processors extremely efficient, and has dramatically reduced the computation cost of garbled-circuit-based protocols.

Garbled circuits are so useful and ubiquitous in cryptography that it has been argued that circuit garbling should be considered as a fundamental cryptographic primitive (like encryption or signatures) rather than as a protocol for two-party secure computation [14].

Key Features: Circuit garbling is inherently a two-party protocol, and requires only a constant number of rounds of communication (independent of the circuit depth). The number of (expensive) public-key operations depends only on the *input size* (OT is a public-key primitive) and the number of private key operations depends on the number of gates. The total communication cost is proportional to the size of the circuit. Since garbled circuit protocols represent each gate by its truth table, the circuit size grows quadratically with the field size when garbling arithmetic circuits. Thus almost all garbled circuits protocols operate on Boolean circuits. There are different methods for garbling arithmetic circuits over large fields [4], but these have never been implemented.

B. Multi-party circuit-based protocols

The GMW [73], BGW [17] and CCD [33] protocols allow an arbitrary number of parties to securely compute a function represented as a circuit. In these protocols, each party uses a linear secret sharing scheme to share its input, and the parties engage in a protocol to compute the answer gate-by-gate. Each gate computation securely transforms secret-shares of the gate's inputs to secret-shares of the gate's outputs. For each addition gate in the circuit, participants can locally compute shares of the output using the linearity of the secret-sharing scheme. Evaluating multiplication gates requires communication, and the schemes differ in how they handle multiplication.

The GMW protocol can evaluate either Boolean or arithmetic circuits, and multiplication gates are executed using Oblivious Transfer for Boolean circuits and with Oblivious Polynomial Evaluation [106] or Oblivious Linear Evaluation [56] for arithmetic circuits. See Ishai *et al.* [85] for a summary of methods for securely computing multiplication gates.

Oblivious communications for multiplication gates dominate the cost of circuit evaluation. All practical GMW-based implementations have taken steps to reduce their overhead. Whether evaluating arithmetic or Boolean circuits, the approach is the same: in an offline pre-computation phase, the participants generate correlated randomness (or receive it from a trusted dealer), and in the online phase, they use these random correlations as masks, or one-time-pads, to compute shares of the output of a multiplication gate based on the shares of the inputs.

Boolean-circuit GMW-based protocols use OT-extension [83] to pre-compute ROT correlations, which are then consumed in the online phase of the protocol. Arithmetic-circuit GMW-based protocols usually generate some form of "Beaver Multiplication Triples" (secret shares of random triples $(a, b, a \cdot b)$, where a, b are field elements) [11] that are used as masks in the online phase.

Information-theoretic protocols, like BGW [17] and CCD [33] rely on secret-sharing schemes supporting *strong multiplication* [43], [35], [31], [44] rather than on public-key primitives. These protocols can be faster, since they do not require computationally expensive public-key operations, but require an honest majority of participants. They generally do not benefit from including a pre-computation phase of the protocol.

Key Features: Multi-party circuit-based protocols can support an arbitrary number of participants. The number

of rounds of communication is proportional to the multiplicative depth of the circuit, and the total amount of communication depends on the number of multiplication gates in the circuit. These protocols allow independent computation parties to receive input and pass output to other parties without compromising security.

C. Hybrid models

Recent systems have moved away from a strict circuit representation and instead use a hybrid model, where functions compile into a set of optimized subprotocols for common operations. This set of primitives, which may include traditional circuit-based operations, are seamlessly described in a single protocol. Hybrid systems often represent intermediate values as secret shares over a large finite field. They may use a mix of information-theoretic and cryptographic protocols, and as such, the number of computation parties and threat models vary.

Hybrid models allow for very different performance characteristics than strict circuit-based models. For example, in a finite field, operations like comparisons, bit-shifts, and equality tests are expensive to represent as an arithmetic circuit. However, specialized sub-protocols that operate on secret shares (e.g., [47], [48], [42]) can compute a sharing of the result far more efficiently.

D. Alternative methods

Fully homomorphic encryption [71] provides an alternative method for securely computing functions encoded as arithmetic circuits. There are several libraries implementing fully homomorphic encryption including HELib [76], PALISADE [118] and SEAL [117]. We do not include these in our comparisons. There are also efforts to garble RAM-model programs [102], [72], [66], instead of circuits, but these have not been implemented.

IV. Frameworks Survey

We survey eleven general-purpose MPC compiler frameworks, all of which follow the same general approach: first, a *compiler* converts a program written in a specialized, high-level language to an intermediate representation (often a circuit). Then the circuit is passed as input to a *runtime*, which executes an MPC protocol and produces an output. We survey two compilers (Frigate and CBMC-GC) that do not have a runtime component.

Table I gives basic information about these frameworks, including protocol type, security settings, availability, and some usability features such as documentation. We limit our scope to recent work: each framework has had a major update since 2014. We do not consider frameworks that are primarily protocol implementations: all the projects included have a developed front-end. We also do not consider standard libraries and APIs related to secure computation. In many cases, we only consider the latest work by a research group.

These frameworks are typically introduced in an academic, peer-reviewed *Paper*. Although many develop a custom or optimized protocol, we group them broadly into *Protocol Families* as described in Section III: Garbled Circuits (GC), Multi-party Circuit protocols (MC), and Hybrid models (Hy). We also note the number of computation *Parties Supported* in a protocol evaluation.

We identify two main threat models: *Semi-honest* adversaries execute the protocol correctly but attempt to glean additional information from the data they receive. A *Malicious* adversary may break the protocol arbitrarily in order to learn information about other inputs or to cause the protocol to output an incorrect result. In practice, security against a malicious adversary is implemented as a "malicious-with-abort" scheme, where the protocol aborts if malicious activity is detected; a malicious adversary cannot cause an incorrect answer, but may result in no answer at all. These descriptors do not apply to frameworks that do not execute a secure computation. We note whether the framework allows *Mixed-Mode* computation: a way to execute both secure and insecure operations in a single program.

In all tables, partial support is indicated by a ◐ symbol; these limitations are explained in detail later in the text.

A. Engineering challenges

Many protocols use a circuit-based model to represent the function being computed. This has the advantage that circuit-based computations are input independent and thus the run-time of the protocol leaks nothing about the user inputs. However, using a circuit model introduces serious challenges and limitations that are present to some degree in all of the frameworks we tested.

Arithmetic circuits operate over a finite field whose size must be set in advance, and must be large enough to prevent overflow (which will vary by application). Arithmetic circuits do not natively support non-arithmetic operations like comparisons and equality checks.

Boolean circuits need to redefine basic operations for every bit width: supporting arithmetic on n-bit integers in such a protocol requires implementing n-bit addition and multiplication circuits. We found no standardization in this area, and most Boolean circuit compilers design and implement their own arithmetic operations. This leads to many restrictions on acceptable programs, and most Boolean-circuit-based frameworks do not support arbitrary bit-width operations.

Compiling high-level programs into circuits requires unrolling all loops and recursive calls. For privacy, the number of loop iterations and recursion depth cannot depend on private inputs. In some situations, static analysis techniques can infer loop termination conditions, but

	Paper	Protocol family	Parties supported	Mixed-mode	Semi-honest	Malicious	Language docs	Online support	Example code	Example docs	Open source	Last major update
EMP-toolkit	[126]	GC	2	●	●	●	○	○	●	○	●	09/2018
Obliv-C	[133]	GC	2	●	●	○	●	○	●	◐	●	06/2018
ObliVM	[101]	GC	2	●	●	○	○	○	●	○	●	02/2016
TinyGarble	[122]	GC	2	○	●	○	○	○	◐	○	◐	10/2018
Wysteria	[115]	MC	2+	●	●	○	◐	○	●	◐	●	10/2014
ABY	[53]	GC,MC	2	●	●	○	●	○	●	●	●	10/2018
SCALE-MAMBA	-	Hy	2+	●	●	●	●	●	●	◐	●	10/2018
Sharemind	[25]	Hy	3	●	●	○	●	●	●	●	◐	09/2018
PICCO	[138]	Hy	3+	●	●	○	●	○	◐	○	●	10/2017
Frigate	[105]	-	2+	○	-	-	●	○	●	○	●	05/2016
CBMC-GC	[82]	-	2+	○	-	-	◐	○	●	○	●	05/2017

TABLE I: A summary of defining features and documentation types. Partial support (◐) is explained in Section V-A.

most compilers do not support such analysis, and instead force the programmer to manually define loop bounds at compile time. Few compilers support recursion.

Conditional operations on secret data can reveal which branch was chosen if the branches take different amounts of computation, so they are typically implemented as a multiplexer, where both branches are evaluated. Similarly, a simple array lookup on a private index must be expanded into a linear-size multiplexer circuit. Frigate, CBMC-GC, and PICCO implement private indexing in this way. Oblivious RAM [74] provides a method for making RAM access patterns data independent, but few frameworks have ORAM support, either natively (ObliVM and SCALE-MAMBA) or via a library (Obliv-C). Most languages do not even allow private array indexing syntactically: if `i` is a "secret" integer and `v` is a "secret" array, then `v[i]` is not valid syntax.

Balancing transparency and flexibility is a key challenge for the MPC compiler designer. MPC protocols often have very different performance characteristics than the corresponding insecure computation, and a compiler that completely hides these differences from the end-user (e.g. automatic multiplexing) in order to provide a more versatile, expressive high-level language can lead developers to write code that is not "MPC-friendly." Alternately, a framework that provides direct access to different back-end representations assumes a high degree of cryptographic expertise on the part of end-users but allows expert users to write highly optimized MPC protocols. It is possible to provide both expressiveness and protocol efficiency without requiring a high degree of cryptographic knowledge on the part of the developer by automatically selecting the optimal MPC protocol for different parts of the code. This type of automatic optimization is difficult, however, and the MPC compilers we analyzed do not attempt it. The EzPC project [32], HyCC framework [30] and ABY[3] project [104], all based on ABY (Section VI-F), attempt to automatically optimize the back-end representation among three protocols. At the time of this writing, none of these projects had code available and we have not included them in our tests.

V. Evaluation Criteria

A. Usability

We consider the tools and documentation a developer needs to install, run, and write programs using the framework. Our findings are summarized in Table I. We identify several types of valuable documentation. Thankfully, every framework we tested includes some form of basic installation documentation. *Language Doc*umentation gives an overview of the high-level language: a language architecture and design document, a start-up guide or tutorial, or a generated list of types and built-in functions. Some larger systems also have *Online Support*, such as an active mailing list or paid personnel who provide technical support[2]. Functional *Example Code* demonstrates end-to-end execution of a program within a framework and is often more up-to-date than general language documentation (a ◐ indicates we needed additional files or tools to run examples). Explicit *Example Doc*umentation provides context for these programs, either in the comments of the code or a separate document (a ◐ indicates limited documentation; details in Section VI).

Most frameworks are *Open Source* under a standard GNU or BSD license (a ◐ indicates closed-source tools or code are required for full functionality). We record the date of the *Last Major Update* (as of this writing): either ongoing development or the latest tagged release.

B. Sample Programs

We implemented three sample programs to evaluate usability, expressiveness, architecture, and cryptographic

[2] Although we received support from academic authors, we do not count responsive authors as "online support;" this model is not scalable.

	Data Types							Operators							Grammar		
	Boolean	Fixed int	Arbitrary int	Float	Array	Dynamic array	Struct	Logical	Comparisons	Addition	Multiplication	Division	Bit-shifts	Bitwise	Conditional	Array access	Private index
EMP-toolkit	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○
Obliv-C	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	Lib
ObliVM	○	●	●	●	◐	●	◐	●	●	●	●	●	●	●	●	●	ORAM
TinyGarble	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Wysteria	●	●	○	○	◐	-	●	○	●	●	●	◐	○	○	●	○	○
ABY	◐	●	○	◐	●	○	●	●	●	●	●	○	○	○	●	◐	○
SCALE-MAMBA	○	●	◐	●	●	○	◐	○	●	●	●	●	●	●	●	●	ORAM
Sharemind	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	○
PICCO	◐	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	Mux
Frigate	○	●	●	○	●	○	●	○	●	●	●	●	●	●	●	●	Mux
CBMC-GC	●	●	○	◐	●	○	●	●	●	●	●	●	●	●	●	●	Mux

TABLE II: A summary of functionality and expressibility of each high-level language. Partial support (◐) is explained for individual frameworks in Section VI.

design of the frameworks. In addition to our online repository, we include illustrative code samples in the Appendix A.

1) Multiply Three: This program takes integer input from three parties and computes the product. The simple function demonstrates the structure of each framework. The program tests whether the implementation supports three or more parties, or if there is a simple way to secret-share multiple inputs across the two computation servers. It also tests basic numeric capabilities of the framework: input and output of integers and simple computation on secure types.

2) Inner Product: The inner product takes the sum of the pairwise product of the elements of two vectors. It tests array-related functionality. Parties should be able to pass an array as input, store secret data within, and access and iterate over the contents. Some frameworks provide ways to parallelize operations over arrays, either through explicit syntactic support or through a parallel architecture device like SIMD gates.

3) Crosstabulation: The crosstabulation program calculates averages by category, where the category table and value table share a primary key but are owned by different parties. This tests framework expressiveness, including input, output, and modification of arrays and conditionals on secret data. In some cases, we tested whether user-defined data types (structs) are supported. We used a simple, brute-force algorithm, and typically returned a list of sums by category (rather than averages).

C. Functionality

We assess the expressive ability of the high-level language used to define secure functions. These criteria are summarized in Table II.

1) Data Types: A fully-supported data type must have both public and secret forms, and the language should allow input and output of the type. These include *Boolean*s, signed or unsigned *Fixed*-length *Int*egers, and more complicated numerical types, such as *Arbitrary*-length *Int*egers and *Float*ing- or fixed-point numbers. Although libraries for these types can be built using fixed-length integer types, we only mark them supported if they are available by default. Combination types include *Array*s and *Dynamic Array*s, where the latter has a size not known at compile time, as well as *Struct*s, user-defined types that can hold other data types as subfields. These complex types are marked supported if they can contain secure data.

2) Operators: Supported operators can be applied to secret data types to get a secret result. We consider *Logical* operators on Booleans and *Comparisons* (equality and inequalities) between integers. We group *Addition* and subtraction as one category, and *Multiplication* and *Division* separately. We consider two bit-level operations: *Bit-shifts* on fixed-length integers and *Bitwise* operators.

*3) Grammar: Conditional*s on a secret Boolean condition can be implemented either with if-statement syntax or a multiplexer operator, though we require an explicit language construct. We consider *Array Access* with a public index and the harder problem of array access with a *Private Index*. The latter can be implemented as a linear-time multiplexer (Mux), native ORAM support (ORAM) or a library for ORAM (Lib).

D. Implementation Criteria

In this section we define architectural and cryptographic criteria, summarized in Table III. We note the main *Development Language* for each framework. Garbled circuit protocols can significantly improve performance by using *AES-NI*, an extension to the x86 instruction set that speeds up AES encryption and decryption.

1) Architecture: We define three broad architecture categories. *Independent* frameworks develop novel lan-

			Architecture			Model			I/O			
	Development language	AES-NI	Independent	Extension	Library	Arithmetic	Boolean	On-the-fly	Arbitrary format	Different input	Array output	Multiple output
EMP-toolkit	C++	●	○	○	●	○	●	●	●	●	◐	●
Obliv-C	OCaml, C	●	○	C	○	○	●	●	●	●	●	●
ObliVM	Java	○	●	○	○	○	●	●	○	●	○	○
TinyGarble	C/C++	●	○	Verilog	○	○	●	●	-	-	-	-
Wysteria	OCaml	-	●	○	○	○	●	-	○	○	-	◐
ABY	C++	●	○	○	●	●	●	○	●	●	●	●
SCALE-MAMBA	Python,C++	-	●	○	○	●	○	-	●	●	◐	●
Sharemind	C/C++	-	●	○	○	●	○	-	●	●	●	●
PICCO	C/C++	-	○	C	○	●	○	-	○	●	●	●
Frigate	C++	-	●	○	○	○	●	○	-	●	●	◐
CBMC-GC	C++	-	○	C	○	○	●	○	○	●	●	◐

TABLE III: Details on architectural and implementation details. Partial support (◐) is explained in Section V-D.

guages and compilers: from limited, domain-specific languages that interface with existing general-purpose languages to stand-alone environments. Some frameworks are *Extension*s of an existing language. These may modify or extend existing compilers to add functionality or take a compiler intermediate state as input. *Library* frameworks are fully implemented in an existing language. They generally define a secure type class and methods for circuit construction and protocol execution.

2) Computation Model: We consider whether the underlying computation model is over an *Arithmetic* field or is based on *Boolean* circuits. Garbled circuit implementations can generate circuits *On-the-Fly*, starting runtime execution before the circuit is fully generated. This can reduce resource consumption, allow dynamic array and loop bounds, and reduce overall program runtime.

3) I/O: Input is typically read from a file, but some frameworks allow *Arbitrary Format*ting, rather than a specific input format. (We've produced input generation scripts for our sample programs.) We note whether the framework supports *Different Input* types from each party. Frameworks should support *Array Output* (a ◐ indicates array elements must be returned one at a time) and *Multiple Output*, where a single party receives two or more output values in a single computation (a ◐ indicates multiple values must be wrapped in a struct).

In Section VI, we comment specifically on restrictions in the I/O file formats, including support for arbitrary-size integers. We recognize that many frameworks are produced in an academic setting that may not value "engineering problems" such as I/O, but we found that the significant usability impact of these capabilities make them worth discussing in this survey.

E. Performance

In this work, we focus on *usability* and do not benchmark framework performance (e.g. run-time, bandwidth, memory-usage, circuit depth). We believe a quantitative evaluation of our sample programs would not accurately represent the performance abilities of each framework. There are several reasons why theoretical efficiency metrics are not always applicable in practical MPC architectures, and we found that direct comparison between different models was often misleading. Circuit size and depth have different implications in garbled circuit and secret-sharing-based protocols, and many frameworks never generate a full circuit for comparison. Execution time varies based on the framework architecture, and preprocessing phases and other variations in execution architecture further complicate timing measurements.

Variations in protocol family and threat model mean that most frameworks are not directly comparable, and our choice of benchmarking function will have a major effect on the relative speed of the frameworks. Our sample programs are designed to reveal the expressive capabilities of a framework and do not necessarily represent a practical MPC use case. A stand-alone measurement for a single run of one of our programs would not take into account the context (typically part of a larger system) in which a secure computation may be evaluated in practice.

We do not wish to disservice incomparable frameworks by providing concrete numbers for impractical test cases. Although a worthwhile and practically useful endeavor, producing a realistic testing framework is beyond the scope of this project.

VI. Frameworks

In this section, we discuss each framework in detail, elaborating on limitations noted in the tables and on the overall usability of each framework. We make recommendations on appropriate use for each framework. We emphasize that many of these frameworks are academic projects, and are therefore subject to the engineering constraints of such an endeavor. Even as we describe the

limitations of these compilers, we wish to emphasize the significant contributions that each has made to the field.

A. EMP toolkit

EMP toolkit [126] is an extensive set of MPC frameworks based on garbled circuits. The core toolkit includes an oblivious transfer library, secure type classes, and several custom protocol implementations. We tested two protocols: a semi-honest implementation of Yao's garbled circuit protocol, and a maliciously secure protocol with authenticated garbling [128]. The toolkit includes three maliciously secure protocols we did not study: a two-party computation that checks input validity [92], a two-party computation library [127], and a multi-party protocol [129].

a) Semi-honest: The implementation of Yao's garbled circuit protocol is a C library defining secure type classes and operations. We found it intuitive for non-expert C developers. The library structure allows developers to use C arrays and structs to hold secure values, and provides simple mixed-mode computation and can generate circuits on-the-fly.

The framework supports arbitrary-size integers and floating-point numbers. Arbitrarily large values can be initialized from a string and returned similarly; they are not restricted to C types. There is little explicit language documentation, but the code was relatively clear. The library can output a protocol-agnostic circuit, but this is not documented.

b) Malicious Authenticated Garbling: This library is primarily an implementation of a custom garbling protocol. We were able to run the included pre-compiled circuit examples and several of our own examples. However, supporting features are limited: functions must be encoded as a circuit prior to computation using the semi-honest library, and I/O is encoded in Boolean arrays.

Recommendation: We recommend the EMP-toolkit semi-honest library for general use. The entire platform is well suited to academics looking to implement a novel circuit-based protocol due to the available circuit generation and cryptographic libraries, but we note that the end-to-end flow is not seamless.

B. Obliv-C

Obliv-C is an extension of C that executes a two-party garbled circuit protocol. The main language addition is an *obliv* qualifier, applied to C types and constructs. Typing rules enforce that *obliv* types remain secret unless explicitly revealed. Code within oblivious functions and conditionals cannot modify public data, except within a qualified ~*obliv* block, in which the code is always executed. These rules allow programmers to reason about data security and develop modular libraries.

The compiler combines these extended functionalities with supporting C code to produce an executable. The executable generates circuits on-the-fly. This allows circuit sizes to depend on values not known during compilation, but may result in under-optimized circuits.

We successfully used an Obliv-C ORAM library, Absentminded Crypto Kit[3], which implements several ORAM variations and other useful primitives [135], [55].

Obliv-C extends C but many of the examples imply an independent-language architecture, separating Obliv-C code from C code. Example programs typically read, process, and output data in native C code, performing only the secure computation in Obliv-C code. However, this abstraction is not enforced: it is possible to perform I/O and call native C functions in Obliv-C files. While many of the examples implement a strict separation between supporting C code and secure Obliv-C code, example documentation uses a mixed paradigm.

Several groups have used Obliv-C to implement secure functionalities, including linear regression [67], decentralized certificate authorities [87], aggregated private machine-learning models [123], classification of encrypted emails [75] and stable matching [54].

Recommendation: Obliv-C is a robust garbled circuit framework. We recommend it to developers for general use and to academics who wish to implement and optimize useful libraries such as ORAM.

C. ObliVM

ObliVM compiles a Java-like language called ObliVM-lang and executes a two-party garbled circuit protocol. It aims to provide a language intuitive to non-experts while implementing domain-specific programming abstractions for improved performance.

ObliVM-lang allows custom data types and type inference. It implements a built-in efficient ORAM scheme. ObliVM natively supports fixed-size integers, and includes a library for arbitrary sized integers.

However, documentation is limited, both for the language (we identified several undocumented reserved keywords), and for general usage. I/O is limited: it requires a non-human-readable format, and we did not find a method to return complex types (including structs and arrays) or more than 32 bits of information. We did not successfully implement the crosstabulation example.

Recommendation: Although ObliVM implements advanced cryptographic constructs, its usability for practical applications is significantly limited by its minimal documentation and restricted I/O functionality.

D. TinyGarble

TinyGarble [122] repurposes hardware circuit generation tools to create optimized circuits appropriate for a garbled circuit protocol. It takes a three-step approach: first, it converts a function defined in Verilog to a *netlist*

[3]https://bitbucket.org/jackdoerner/absentminded-crypto-kit

format. Then it converts the netlist format to a custom circuit description (SCD) and securely evaluates the Boolean circuit using a garbled circuit protocol.

We found that the first step of this process requires a closed-source logic synthesis tool (the Synopsys Design Compiler) that converts a Verilog file to the unstandardized netlist format. The authors reference an open-source tool, Yosys Open SYnthesis Suite, but we were unable to compile any examples (conversions from Yosys-produced netlist files to SCD failed). The source code for TinyGarble includes some pre-compiled netlist files. While we could see every step for these examples (Verilog source, netlist file, computation output), we were unable to compile examples end-to-end and thus do not make any claims about language functionality.

TinyGarble is preceded by JustGarble [13], a library for circuit garbling and evaluation. JustGarble does not include communication or circuit generation. The garbled circuit implementation in TinyGarble is a strict improvement over JustGarble, including recent protocol and circuit optimizations. A follow-up to TinyGarble [51] leverages hardware logic synthesis tools to optimze for GMW-style computations.

Recommendation: TinyGarble aims to leverage powerful circuit optimizers developed for producing hardware circuits. Unfortunately, from a usability standpoint, the lack of compatibility for Verilog compilers and lack of standards around netlist formats meant that we were unable to compile or run any new examples using the TinyGarble framework. We believe, however, that the MPC community could benefit greatly by leveraging the power of existing circuit optimizers.

E. Wysteria

Wysteria [115] introduces a novel high-level *functional* programming language. It guarantees that a distributed secure computation produces the same output as a single trusted party. Wysteria supports an arbitrary number of computation parties, and the software contribution includes a front-end language, a type checker, and a run-time interpreter that executes a Boolean-circuit-based GMW protocol implementation [39].

Wysteria supports mixed-mode computation via a language construct called a *secure block*. A secure block is initialized with a set of parties and their inputs. All operations in the block's scope are compiled to a Boolean circuit and executed as a separate computation.

The Wysteria codebase has changed since the original publication of the paper, and the examples presented in the paper do not compile. However, they provide useful context for the architecture of a working program. The repository includes several example programs that run without errors, including a 6-party version of the millionaire's problem.

Wysteria includes a record type, which holds named values of other types and can be returned from a secure block. Although Wysteria includes working examples that input an array to a secure block, we were unable to replicate this for our inner product or crosstabulation programs. The language has support for iterating over arrays in secure blocks, and allows access to individual array elements outside of a secure block. We encountered other significant language limitations: Wysteria only supports division by 2 in secure blocks, and we did not find a way to use logical operators on Booleans.

The Wys* project [116] built on the ideas of Wysteria, and attempted to create a fully-verified toolchain for secure computation based on the F* programming language. The F* language is undergoing rapid development, however, and we were not able to compile Wys*.

Recommendation: Wysteria's limited support for complex data types, current lack of development, and outdated back-end circuit parser, mean that it should not be used for developing complex or efficient protocols. On the other hand, Wysteria is the only compiler we examined that intends to provide a system for automatically verifying that the underlying multi-party computation has the same functionality as the monolithic program implemented by the developer, and the only compiler with a functional-style programming language. We recommend future compiler developers use Wysteria's type-based correctness and security guarantees as a model.

F. ABY

ABY [53] is a mixed-protocol two-party computation framework implemented as a C++ library. It aims to give developers fine-grained control over computation efficiency by providing a mechanism for mixing protocols. ABY switches between three protocols: The GMW-based *Arithmetic* protocol uses an additive sharing scheme with multiplicative triples on an arithmetic circuit. The protocol is based on those by [8], [94], [113]. The other protocols use Boolean circuits: the *Boolean* protocol implements the original GMW protocol with an XOR-based sharing scheme, while the *Yao* protocol uses an optimized version of Yao's garbled circuit protocol.

ABY has a significant amount of documentation that provides a helpful framework for understanding the capabilities of the framework. This includes a slightly outdated developer guide, an extended README file, and a variety of commented examples.

Secure data is limited to unsigned C integer types: ABY does not support arbitrary-length integers or an explicit Boolean type, although it allows one-bit integers that function equivalently. It supports some floating-point operations and is actively developing this functionality. ABY can store secure data in a C struct, and supports both C++ arrays and SIMD constructions for efficient

parallel operations. ABY provides functions for creating and populating SIMD "shares", but retrieving individual elements requires operating on the internal representation of secret data, which is not well-supported.

ABY has been used to implement several secure computation systems [3], [36], [37], [52], [96], [112].

Recommendation: ABY provides a powerful, low-level cryptographic interface that gives the developer significant control over performance. ABY is targeted at users who are familiar with MPC protocols and the circuit model of computation. We recommend it to developers with sufficient cryptographic background.

G. SCALE-MAMBA

SCALE-MAMBA implements a maliciously secure two-phase hybrid protocol and supersedes the SPDZ framework. MAMBA is a Python-like language that compiles to a bytecode representation. SCALE implements a two-phase protocol which offloads all public-key operations to an offline pre-processing phase, generates three types of shared randomness to use during protocol execution, and executes an optimized hybrid protocol based on previous work [19], [49], [107].

SCALE-MAMBA has a significant amount of documentation, covering the differences from the previous SPDZ system, installation and runtime instructions, updated language documentation, and protocol primitives. The example programs are unit-style tests but are not explicitly documented. A community bulletin board[4] hosts discussion and questions about the framework.

The SCALE-MAMBA framework allows developers to define their own I/O classes. This provides an extremely flexible interface. We did not implement a custom I/O class. The framework's secure channel setup requires users to produce a mini certificate authority in order to run a computation.

Running our sample programs with a simple full-threshold secret sharing scheme required significant memory resources. However, the system offers multiple, customizable options for a secret-sharing scheme, and includes programmatic tools for offline data generation in certain contexts. For testing purposes, SCALE includes an option to run with fake (insecure) offline data.

Integer size is determined by the field size, which must be chosen at compile time. Standard full-threshold sharing supports a modulus of up to 1024 bits. SCALE-MAMBA supports most bit-shift operations and includes Python-style tuples, which we consider to be a less powerful type of struct. Fixed-point numbers are fully supported, and floating-point numbers partially implemented. It has ORAM support, which we did not test.

Recommendation: We recommend SCALE-MAMBA for a variety of uses: it is flexible, supports an arbitrary number of parties and has strong security guarantees, though it may require significant computing resources.

[4]https://groups.google.com/forum/#forum/spdz

H. Sharemind MPC

Sharemind [25] is a secure data processing platform and a trademark of Cybernetica, a research-and-development-focused technology company based in Tallinn, Estonia. We used the Sharemind MPC platform, which securely executes a function written in the SecreC language. The framework executes a three-party hybrid protocol using an additive secret sharing scheme.

The Sharemind MPC platform explicitly defines three parties: *clients*, who input values; *servers*, who define and run the secure computation; and *outputs*, who receive the output of the computation. Server code is written in the SecreC language and executed using Sharemind's secure runtime, while client and output code uses a client library in a common programming language and is executed using standard compilers. We developed our sample programs using a C client library; the platform also provides libraries in Haskell, Java, and JavaScript.

Sharemind MPC implements a custom additive secret-sharing scheme over a fixed-size ring. These fixed-size integers have behavior consistent with traditional C integers, and the framework includes a floating point library. The protocol is written for exactly three servers, but there is support for arbitrarily many parties secret-sharing their inputs among the three computation servers into a database structure. Our samples passed all input values from a single client program. The SecreC language is expressive and well documented online[5]. The supporting client libraries are not as well documented.

The Sharemind MPC platform has several licensing options[6] through Cybernetica. We used the academic server. This license gave us access to the protocol implementation. The platform also includes an open-source simulator, which includes the SecreC language, its standard library, and an emulator for the secure computation. The emulator is available as a VM and as compilable source code; we successfully ran our examples on both versions. The emulator does not support client code and arguments must be passed on the command line.

As part of the academic license, we had access to several Sharemind employees who provided "reasonable assistance" throughout the development process, and we consider this to be an *online resource*.

Recommendation: The Sharemind MPC platform is suitable for a wide variety of purposes. We recommend it to companies looking to implement secure computation, particularly for large or complex functionalities, as well as to academics who require MPC as a tool for a project.

[5]https://sharemind-sdk.github.io/stdlib/reference/index.html
[6]https://sharemind.cyber.ee/sharemind-mpc/

I. PICCO

PICCO [138], [137] is a general-purpose compiler with a custom secret-sharing protocol. It includes three main software contributions: a compiler that translates an extension of C to a native C implementation of the secure computation; an I/O utility that produces and reconstructs secret-shared input and output; and a tool that initiates the computation. PICCO executes computations under a hybrid model, using an information-theoretic protocol for multiplication [70] and custom primitives for other operations. It supports an arbitrary number of parties but requires an honest majority.

PICCO allows conditionals on private variables, but does not allow public variables to be assigned within the scope of such a conditional. It also allows indexing arrays at a private location, though this is implemented as a multiplexer, not an ORAM scheme. It supports pointers to private data and dynamic memory allocation using standard C-like syntax. The language supports single-bit integer types to approximate Booleans and while we ran provided examples, we were unable to successfully write our own program that uses them.

The language is well-documented in the paper. The code repository includes many examples in a C extension, but doesn't include examples of the additional files needed to compile and run a program end-to-end. The process to compile and execute a secure computation is lengthy but well-documented and requires multiple configuration files and explicit generation and reconstruction of secret-shared inputs and outputs.

Recommendation: PICCO is appropriate for developers or academics who require a true multi-party implementation. We found no correctness issues and it allows a great deal of flexibility in configuring the computation.

J. Frigate

Frigate [105] compiles a novel C-like language to a custom Boolean circuit representation for any number of inputs. The framework emphasizes the use of good software engineering techniques, including an extensive testing suite and a focus on modularity and extensibility. The circuit format minimizes file size, and the framework includes an interpreter to efficiently interface between generated circuits and other applications.

Frigate produces a circuit, so all operations are secure by default. The type system is extremely simple, with only three native types: signed and unsigned integers and structs. While there is no explicit Boolean type, integers are of arbitrary size and the language defines comparison and bitwise operators, so it supports equivalent functionality. Global variables are not allowed. Frigate allows arrays but they must be contained within structs. The circuit compiler provides useful errors, and the source code includes a brief description of interpreter options and a language description.

One usability issue is that basic arithmetic operations are defined only for operands of the same type and size. This may increase circuit size for some applications. The framework does not include a simulator, so any correctness checking requires a separate back end. To test the circuits generated by Frigate, we wrote a tool that converts Frigate circuits into a format suitable for execution in an implementation of the BMR protocol [16].

Recommendation: Frigate provides an expressive C-like language for fast circuit generation and is a good way to estimate the circuit size of a given computation. However, even with our conversion tools that connect Frigate's circuit form to useful back-ends, executing an end-to-end MPC computation requires relatively burdensome action from the user.

K. CBMC-GC

CBMC-GC [82], [64] produces Boolean circuits from a subset of ANSI-C. It is based on CMBC [40], a bounded model checker that translates any C program into a Boolean constraint then adapts the output of this tool to produce an optimized circuit for an MPC computation. The compiler can optimize for minimal size or minimal depth circuits. It produces circuits for any number of input parties; we compiled and simulated sample programs with up to ten parties.

We did not find adequate documentation for the limitations of the adapted subset of ANSI-C that CMBC-GC compiles. For example, variable names for inputs to the main file must be prefixed by `INPUT_`. Arrays cannot be passed natively as arguments; they must be wrapped in a struct. Non-default integer types, such as specific-width integers, can be used but need to be explicitly included. We were unable to compile a program using C Boolean types. The framework includes a rudimentary set of floating-point operations, and allows conditionals on secret data. There are some configuration options, such as circuit optimization technique, depth to unroll loops, and a time limit on minimization.

CBMC-GC includes a tool for running circuits with ABY (Section VI-F). We were unable to run an example with this converter; it appears that the CBMC-GC code references a deprecated ABY API. CBMC-GC also includes a tool to output circuits in other formats, including the Simple Circuit Description (SCD) used by the TinyGarble compiler; Fairplay's Secure Hardware Definition Language (SHDL); and the Bristol circuit format [124]. We tested the output of this tool with TinyGarble's compiler (Section VI-D), but were unable to run any examples; we weren't able to determine whether the errors were due to circuit generation by CBMC-GC or execution by TinyGarble.

Recommendation: CBMC-GC uses powerful tools to produce optimized circuits, but we were not able to successfully execute any of the circuits it produced.

VII. Discussion

A. Leveraging Existing Compiler Research

Programming language research is a field dedicated to creating compilers but little MPC research leverages these techniques. Wysteria is a notable exception, but it has significant engineering gaps that make it unusable for practical computations. However, the MPC community would benefit if frameworks took a more principled approach to language design and verification.

One notable area for improvement is compiler correctness. We found that while the frameworks were generally successful in preventing security mistakes, many had correctness issues. Defining and implementing type rules that guarantee a correct output could reduce these issues, which were often silent failures.

B. Documentation

Universally, the biggest obstacle when using MPC frameworks was a lack of documentation. The community has put thousands of hours into producing the work presented herein, and even mediocre documentation makes these contributions significantly more accessible.

Documentation comes in many forms and having multiple types of documentation is helpful when using a complex software system such as these. Our evaluation criteria suggest several types we found particularly useful, and we encourage developers and researchers to produce multiple resources for system users.

In addition to static documentation provided by the authors, active online resources can be extremely valuable. These include archived correspondence, like an archived mailing list, Google group or issue tracker on GitHub. These resources can reduce the burden on researchers who may be asked the same (or similar) questions repeatedly via private correspondence. Example programs are also an important resource, and a repository where the community can archive simple example programs (e.g. like http://www.texample.net) would dramatically improve usability and utility of these systems.

C. Standardization and Benchmarks

Many of these frameworks are designed around a particular feature, such as a type system or an optimization technique. Standardizing essential features common across frameworks allows researchers to concentrate their efforts on core features of their systems and provides a level of consistency for users. Standardization could also set a more consistent baseline for performance measurements. One potential issue is standardizing on a soon-to-be-obsolete technology. For example, while we could recommend a circuit format, this would not be useful for modern hybrid framework models that use a different intermediate representation.

Several projects are developing standardization in the field. SCAPI [1], [61] defines a general API that provides a common interface for cryptographic building blocks and primitives commonly used in secure computation. It aims to provide a uniform, flexible, and efficient standard library for cryptographers to use in their MPC implementations and includes significant documentation. FRESCO [65] defines a set of Java APIs for function description and protocol definition and evaluation. As a demonstration, the project includes front-end code for several sample projects and a new implementation of the SPDZ protocol with MASCOT preprocessing. The SCALE-MAMBA systems use a set of bytecodes as an intermediate representation that have been reused in other projects, such as the Jana compiler [5].

Benchmarking performance across frameworks presents a challenge due to the variety of dependencies on processing power, network bandwidth, network latency, computation structure, and framework architecture. Theoretical performance measures can be difficult to measure in practice and frameworks that excel in one benchmark environment may fare poorly in another. Nevertheless, benchmarks can provide insight into a framework's strengths and weaknesses, and do have value if they are not used as a sole measure of a framework's contribution. Recent work by Barak *et al.* [9] provides an approach for performance comparison between frameworks with compatible architectures.

We recommend that the community collectively develops a consistent set of problems and associated metrics that demonstrate the expressive capabilities of a framework and serve as a baseline for performance comparison. Standardized benchmarking has some known issues: certain metrics may not be relevant to every protocol; compilers may optimize for performance on benchmark problems rather than in the general case; and the performance measurement issues from Section V-E remain. We hope that careful design of benchmarking problems will mitigate these issues and provide a useful tool for practitioners in the future.

Acknowledgements

The authors would like to thank the developers and maintainers of each framework for their help and feedback, and the anonymous reviewers for their helpful comments. This research was sponsored in part by ONR grant (N00014-15-1-2750) "SynCrypt: Automated Synthesis of Cryptographic Constructions" and NSF grant CNS-1513671.

REFERENCES

[1] libscapi. https://github.com/cryptobiu/libscapi. Accessed 25 June 2018.

[2] Emmanuel A. Abbe, Amir E. Khandani, and Andrew W. Lo. Privacy-Preserving Methods for Sharing Financial Risk Exposures. *American Economic Review*, 102(3):65–70, May 2012.

[3] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454. Springer, 2015.

[4] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. *SIAM Journal on Computing*, 43(2):905–929, 2014.

[5] David W. Archer, Dan Bogdanov, Y. 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. Cryptology ePrint Archive, Report 2018/450, 2018. https://eprint.iacr.org/2018/450.

[6] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. Cryptology ePrint Archive, Report 2015/1039, 2015. https://eprint.iacr.org/2015/1039.

[7] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *20. ACM Conference on Computer and Communications Security (CCS'13)*, pages 535–548. ACM, 2013. Full version: https://ia.cr/2013/552. Code: https://encrypto.de/code/OTExtension.

[8] Mikhail Atallah, Marina Bykova, Jiangtao Li, Keith Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 103–114, New York, NY, USA, 2004. ACM.

[9] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 695–712. ACM, 2018.

[10] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC '90*, pages 503–513, New York, NY, USA, 1990. ACM.

[11] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[12] Amos Beimel. Secret-Sharing Schemes: A Survey. In Yeow-Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology*, volume 6639 of *Lecture Notes in Computer Science*, pages 11–46. Springer Berlin Heidelberg, 2011.

[13] M. Bellare, Viet T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, May 2013.

[14] Mihir Bellare, Viet T. Hoang, and Phillip Rogaway. Foundations of Garbled Circuits. In *CCS '12*, CCS '12, pages 784–796, New York, NY, USA, 2012. ACM.

[15] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A system for secure multi-party computation. In *CCS '08*, pages 257–266, 2008.

[16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *CCS '16*, pages 578–590, New York, NY, USA, 2016. ACM.

[17] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, New York, NY, USA, 1988. ACM.

[18] Josh Benaloh and Jerry Leichter. Generalized Secret Sharing and Monotone Functions. In Shafi Goldwasser, editor, *Advances in Cryptology CRYPTO' 88*, volume 403 of *Lecture Notes in Computer Science*, pages 27–35. Springer New York, 1988.

[19] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT '11*, pages 169–188, 2011.

[20] Michael Bertilsson and Ingemar Ingemarsson. A construction of practical secret sharing schemes using linear block codes. In Jennifer Seberry and Yuliang Zheng, editors, *Advances in Cryptology AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 67–79. Springer Berlin Heidelberg, 1993.

[21] G. R. Blakley. Safeguarding cryptographic keys. In *Managing Requirements Knowledge, International Workshop on*, volume 0, page 313, Los Alamitos, CA, USA, December 1979. IEEE Computer Society.

[22] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proceedings on Privacy Enhancing Technologies*, 3:117–135, 2016.

[23] Dan Bogdanov, Liina Kamm, Sven Laur, Pille Pruulmann-Vengerfeldt, Riivo Talviste, and Jan Willemson. Privacy-Preserving Statistical Data Analysis on Federated Databases. In Bart Preneel and Demosthenes Ikonomou, editors, *Privacy Technologies and Policy*, volume 8450 of *Lecture Notes in Computer Science*, pages 30–55. Springer International Publishing, 2014.

[24] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, June 2014.

[25] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, volume 5283 of *ESORICS '08*, pages 192–206, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, September 2012.

[27] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying Secure Multi-Party Computation for Financial Data Analysis. In AngelosD Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer Berlin Heidelberg, 2012.

[28] Ernest F. Brickell. Some Ideal Secret Sharing Schemes. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology EUROCRYPT '89*, volume 434 of *Lecture Notes in Computer Science*, chapter 45, pages 468–475. Springer Berlin Heidelberg, Berlin, Heidelberg, May 1989.

[29] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 15, Berkeley, CA, USA, 2010. USENIX Association.

[30] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *25. ACM Conference on Computer and Communications Security (CCS'18)*, pages 847–861. ACM, 2018.

[31] Ignacio Cascudo, Hao Chen, Ronald Cramer, and Chaoping Xing. Asymptotically good ideal linear secret sharing with strong multiplication over any fixed finite field. In *Advances in Cryptology-CRYPTO 2009*, pages 466–486. Springer, 2009.

[32] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation. IACR ePrint Archive 2017/1109, 2017.

[33] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty Unconditionally Secure Protocols. In *STOC*, pages 11–19, 1988.

[34] Erika Check Hayden. Extreme cryptography paves way to personalized medicine. *Nature*, 519(7544):400–401, March 2015.

[35] Hao Chen and Ronald Cramer. Algebraic Geometric Secret Sharing Schemes and Secure Multi-Party Computations over Small Fields. In *CRYPTO '06*, pages 521–536, 2006.

[36] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. Towards securing internet exchange points against curious onlookers. In *ANRW*, pages 32–34, New York, NY, USA, 2016. ACM.

[37] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. SIXPACK: Securing Internet eXchange Points Against Curious onlooKers. In *13. International Conference on emerging Networking EXperiments and Technologies (CoNEXT'17)*, pages 120–133. ACM, 2017.

[38] Hyunghoon Cho, David J Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology*, 36(6):547, 2018.

[39] Seung G. Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 416–432, February 2012.

[40] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.

[41] Martine de Cock, Rafael Dowsley, Anderson CA Nascimento, and Stacey C Newman. Fast, privacy preserving linear regression over distributed datasets based on pre-distributed data. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2015.

[42] Geoffroy Couteau. New protocols for secure equality test and comparison. In *International Conference on Applied Cryptography and Network Security*, pages 303–320. Springer, 2018.

[43] Ronald Cramer, Ivan Damgård, and Ueli Maurer. General Secure Multi-party Computation from any Linear Secret-Sharing Scheme. In Bart Preneel, editor, *Advances in Cryptology EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, chapter 22, pages 316–334. Springer Berlin Heidelberg, Berlin, Heidelberg, May 2000.

[44] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. *Secure Multiparty Computation and Secret Sharing An Information Theoretic Approach*. Cambridge University Press, 2015.

[45] Ronald Cramer and Serge Fehr. Optimal Black-Box Secret Sharing over Arbitrary Abelian Groups. In Moti Yung, editor, *Advances in Cryptology CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, chapter 18, pages 272–287. Springer Berlin Heidelberg, Berlin, Heidelberg, September 2002.

[46] Claude Crépeau. Equivalence Between Two Flavours of Oblivious Transfers. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 350–354, London, UK, UK, 1988. Springer-Verlag.

[47] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multiparty computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, pages 285–304. Springer, 2006.

[48] Ivan Damgard, Martin Geisler, and Mikkel Kroigard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1(1):22–31, 2008.

[49] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[50] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper B. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *PKC '09*, volume 5443 of *PKC 2009, Lecture Notes in Computer Science*, pages 160–179. Springer, 2009. See http://viff.dk.

[51] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In *22. ACM Conference on Computer and Communications Security (CCS'15)*, pages 1504–1517. ACM, 2015.

[52] Daniel Demmler, Kay Hamacher, Thomas Schneider, and Sebastian Stammler. Privacy-preserving whole-genome variant queries. In *16. International Conference on Cryptology And Network Security (CANS'17)*, volume 11261 of *LNCS*, pages 71–92. Springer, 2017.

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

[54] Jack Doerner, David Evans, and abhi shelat. Secure stable matching at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1602–1613. ACM, 2016.

[55] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 523–535, New York, NY, USA, 2017. ACM.

[56] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2263–2276. ACM, 2017.

[57] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative scientific computations. In *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*, pages 273–282. IEEE, 2001.

[58] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative statistical analysis. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual*, pages 102–110. IEEE, December 2001.

[59] Wenliang Du and Mikhail J. Atallah. Secure multi-party computation problems and their applications: a review and open problems. In *Proceedings of the 2001 workshop on new security paradigms*, NSPW '01, pages 13–22, New York, NY, USA, 2001. ACM.

[60] Wenliang Du, Yunghsiang S. Han, and Shigang Chen. Privacy-Preserving Multivariate Statistical Analysis: Linear Regression and Classification. In *In Proceedings of the 4th SIAM International Conference on Data Mining*, pages 222–233, 2004.

[61] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: the secure computation application programming interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.

[62] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[63] Mark Flood, Jonathan Katz, Stephen Ong, and Adam Smith. Cryptography and the Economics of Supervisory Information: Balancing Transparency and Condentiality. Technical Report 0011, Office of Financial Research, September 2013.

[64] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In Albert Cohen, editor, *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 244–249. Springer Berlin Heidelberg, 2014.

[65] Alexandra Institute FRESCO Team. Framework for efficient secure computation (FRESCO). https://github.com/aicis/fresco, 2018.

[66] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In *FOCS '15*, pages 210–229. IEEE, 2015.

[67] Adria Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans.

Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies*, 4:248–267, 2017.

[68] Martin Geisler, Tomas Toft, Mikkel Krigrd, Thomas Pelle Jakobsen, Jakob Illeborg Pagter, Sigurd Meldgaard, Marcel Keller, Tord Reistad, Ivan Damgrd, and Janus Dam Nielsen and. VIFF. http://viff.dk.

[69] Rosario Gennaro. *Theory and Practice of Verifiable Secret Sharing*. PhD thesis, MIT, 1996.

[70] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 101–111, New York, NY, USA, 1998. ACM.

[71] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09*, pages 169–178, New York, NY, USA, 2009. ACM.

[72] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *EUROCRYPT*, pages 405–422. Springer, 2014.

[73] Oded Goldreich, Sylvio Micali, and Avi Wigderson. How to Play any Mental Game. In *STOC '87*, pages 218–229, 1987.

[74] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[75] Trinabh Gupta, Henrique Fingler, Lorenzo Alvisi, and Michael Walfish. Pretzel: Email encryption and provider-supplied functions are compatible. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 169–182. ACM, 2017.

[76] Shai Halevi and Victor Shoup. Helib: Implementing homomorphic encryption. http://shaih.github.io/HElib/index.html, 2015.

[77] Danny Harnik, Yuval Ishai, Eyal Kushilevitz, and Jesper Buus Nielsen. OT-combiners via secure computation. In *Theory of Cryptography Conference*, pages 393–411. Springer, 2008.

[78] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. High-precision secure computation of satellite collision probabilities. In *SCN*, pages 169–187. Springer, 2016.

[79] Brett Hemenway, William Welser, and Dave Baiocchi. Achieving Higher-Fidelity Conjunction Analyses Using Cryptography to Improve Information Sharing. Technical report, RAND Corporation, 2014.

[80] Brian Hie, Hyunghoon Cho, and Bonnie Berger. Realizing private and practical pharmacological collaboration. *Science*, 362(6412):347–350, 2018.

[81] Martin Hirt and Ueli Maurer. Player Simulation and General Adversary Structures in Perfect Multiparty Computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[82] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure Two-party Computations in ANSI C. CCS '12, pages 772–783, New York, NY, USA, 2012. ACM.

[83] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, pages 145–161, 2003.

[84] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding Cryptography on Oblivious Transfer - Efficiently. In *CRYPTO '08*, pages 572–591, 2008.

[85] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure Arithmetic Computation with No Honest Majority. In *TCC*, pages 294–314, March 2009.

[86] Karthik A. Jagadeesh, David J. Wu, Johannes A. Birgmeier, Dan Boneh, and Gill Bejerano. Deriving genomic diagnoses without revealing patient genomes. *Science*, 357(6352):692–695, 2017.

[87] Bargav Jayaraman, Haina Li, and David Evans. Decentralized certificate authorities. arXiv preprint arXiv:1706.03370, 2017.

[88] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 216–230. IEEE, 2008.

[89] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, April 2013.

[90] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, pages 1–18, 2014.

[91] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. Privacy-preserving analysis of vertically partitioned data using secure matrix products. *Journal of Official Statistics*, 25(1):125, 2009.

[92] Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. https://eprint.iacr.org/2016/184.

[93] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *Annual Cryptology Conference*, pages 724–741. Springer, 2015.

[94] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, pages 566–584, Cham, 2014. Springer International Publishing.

[95] Joe Kilian. Founding cryptography on oblivious transfer. In *STOC '88*, STOC '88, pages 20–31, New York, NY, USA, 1988. ACM.

[96] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies*, 2017(4):177–197, 2017.

[97] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 486–498, Berlin, Heidelberg, 2008. Springer-Verlag.

[98] Enrique Larraia. Extending oblivious transfer efficiently. In *International Conference on Cryptology and Information Security in Latin America*, pages 368–386. Springer, 2014.

[99] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[100] Yehuda Lindell and Benny Pinkas. Secure Multiparty Computation for Privacy-Preserving Data Mining. *The Journal of Privacy and Confidentiality*, 1(1):59–98, 2009.

[101] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 359–376, Washington, DC, USA, 2015. IEEE Computer Society.

[102] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, pages 719–734. Springer, 2013.

[103] Dahlia Malkhi, Noan Nisan, Benny Pinkas, and Yaron Sella. Fairplay – A Secure Two-Party Computation System. In *USENIX Security Symposium '04*, 2004.

[104] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 35–52, New York, NY, USA, 2018. ACM.

[105] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (Euro S&P)*, pages 112–127, March 2016.

[106] Moni Naor and Benny Pinkas. Oblivious polynomial evaluation. *SIAM Journal on Computing*, 35(5):1254–1281, 2006.

[107] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 681–700. Springer, 2012.

[108] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and*

Privacy (SP), 2013 IEEE Symposium on, pages 334–348. IEEE, 2013.

[109] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N Wright. Ssc protocol comparison tool. https://code.google.com/archive/p/syssc-ui/. Accessed 2018-06-25.

[110] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N Wright. Ssc protocol comparison tool. http://work.debayangupta.com/ssc/. Accessed 2018-06-25.

[111] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N Wright. Systematizing secure computation for research and decision support. In *International Conference on Security and Cryptography for Networks*, pages 380–397. Springer, 2014.

[112] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *EUROCRYPT*, pages 125–157. Springer, 2018.

[113] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. The design and implementation of a two-party protocol suite for Sharemind 3. Cybernetica Research Reports T-4-17, 2012.

[114] Michael Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University, 1981.

[115] Aseem Rastogi, Matthew Hammer, and Michael Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 655–670. IEEE, May 2014.

[116] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. Wys*: A verified language extension for secure multi-party computations. arXiv preprint arXiv:1711.06467, 2017.

[117] Microsoft Research. Simple encrypted arithmetic library (SEAL). https://www.microsoft.com/en-us/research/project/simple-encrypted-arithmetic-library/, 2016.

[118] Kurt Rohloff and Gerard Ryan. The PALISADE lattice cryptography library. https://git.njit.edu/palisade/PALISADE, 2016.

[119] Drago Rotaru. awesome-mpc. https://github.com/rdragos/awesome-mpc, 2018.

[120] Adi Shamir. How to share a secret. *Commun. ACM*, 22:612–613, November 1979.

[121] Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. Practical secure computation outsourcing: A survey. *ACM Computing Surveys (CSUR)*, 51(2):31, 2018.

[122] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE S & P*, 2015.

[123] Lu Tian, Bargav Jayaraman, Quanquan Gu, and David Evans. Aggregating private sparse learning models using multi-party computation. In *NIPS Workshop on Private Multi-Party Machine Learning, Barcelona, Spain*, 2016.

[124] Stefan Tillich and Nigel Smart. Circuits of basic functions suitable for mpc and fhe. https://homes.esat.kuleuven.be/%7Ensmart/MPC. Accessed 21 June 2018.

[125] Marten Van Dijk. *Secret key sharing and secret key generation*. PhD thesis, Eindhoven University of Technology, 1997.

[126] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[127] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, pages 399–424, 2017.

[128] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 21–37, New York, NY, USA, 2017. ACM.

[129] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 39–56, New York, NY, USA, 2017. ACM.

[130] Stephen Wiesner. Conjugate coding. *SIGACT News*, 15(1):78–88, January 1983.

[131] Andrew Yao. Protocols for Secure Computations (Extended Abstract). In *FOCS '82*, pages 160–164, 1982.

[132] Andrew Yao. How to Generate and Exchange Secrets. In *FOCS '86*, pages 162–167, 1986.

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

[134] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

[135] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root oram: efficient random access in multi-party computation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 218–234. IEEE, 2016.

[136] Nicholette Zeliadt. Cryptographic methods enable analyses without privacy breaches. *Nature Medicine*, 20(6):563, June 2014.

[137] Yihua Zhang, Marina Blanton, and Ghada Almashaqbeh. Implementing support for pointers to private data in a general-purpose secure multi-party compiler. *ACM Trans. Priv. Secur.*, 21(2):6:1–6:34, December 2017.

[138] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A general-purpose compiler for private distributed computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 813–826, New York, NY, USA, 2013. ACM.

APPENDIX A
INNER PRODUCT CODE

Here we include snippets of code that implement the inner product function in each language. They have been slightly modified from the versions on Github for length.

a) EMP-toolkit: Examples are contained in a single file. They are compiled to a single executable using CMake, which is executed by both parties.

```
void innerprod(int bits,
        string inputs_a[],
        string inputs_b[], int len) {
  Integer sum(bits, 0, PUBLIC);
  for( int i=0; i<len; i++) {
    Integer a(bits, inputs_a[i], ALICE);
    Integer b(bits, inputs_b[i], BOB);
    sum = sum + (a * b);
  }
  cout << sum.reveal<int>() << endl;
}
```

b) Obliv-C: Examples are often split into two parts: C header and implementation files that read input and initialize network connections, and an Obliv-C file that defines the secure computation. Examples are compiled with a simple `Makefile`.

```
protocolIO io = args;
int len = ocBroadcastInt(io->input.size,1);
obliv int* v1 =
  malloc(sizeof(obliv int) * len);
obliv int* v2 =
  malloc(sizeof(obliv int) * len);
feedOblivIntArray(v1,
  io->input.arr, len, 1);
feedOblivIntArray(v2,
  io->input.arr, len, 2);
```

```
obliv int sum = 0;
for(int i = 0; i < len; i++){
    sum += v1[i] * v2[i];
}
revealOblivInt(&(io->result), sum, 0);
```

c) ObliVM: Examples are a single file defining the secure computation. They are compiled and run using several short scripts.

```
int main@n@m(int@n x, int@m y){
  secure int32[public (n/32)] a;
  secure int32[public (m/32)] b;
  public int32 len = n/32;
  for(public int32 i=0; i<len; i=i+1){
    a[i] = x$32i~32*(i+1)$;
    b[i] = y$32i~32*(i+1)$;
  }
  secure int32 sum = 0;
  for(public int32 i=0; i<len; i=i+1){
    sum = sum + (a[i] * b[i]);
  }
  return sum;
}
```

d) TinyGarble: We were not able to produce working examples for TinyGarble.

e) Wysteria: We were not able to write a working example for inner product in Wysteria; instead, we include code to multiply 3 numbers. Wysteria examples are contained in a single file. Each party runs the type checker and executes the computation.

```
let parts = { !Alice, !Bob, !Charlie } in
let result @par( parts ) =
  let w = wire { !Alice }:5 in
  let w = (wire { !Bob }:2) ++ w in
  let w = (wire { !Charlie }:1) ++ w in
  let product @sec( parts ) =
    (w[!Alice] * w[!Bob] ) * w[!Charlie]
  in product
in wire { parts }:result
```

f) ABY: Examples are typically structured as several C++ files: one that reads input and configuration values and header and implementation files that define and execute the secure computation. They are compiled and executed with a complex Make system.

```
share BuildInnerProductCircuit(
  share *s_x, share *s_y, uint32_t num,
  ArithmeticCircuit ac) {
  uint32_t i;
  s_x = ac->PutMULGate(s_x, s_y);
  s_x = ac->PutSplitterGate(s_x);
  for (i = 1; i < num; i++) {
    s_x->set_wire_id(0,
      ac->PutADDGate(s_x->get_wire_id(0),
      s_x->get_wire_id(i)));
  }
  s_x->set_bitlength(1);
  s_out = circ->PutOUTGate(s_x, ALL);
  return s_out;
}
```

g) SCALE-MAMBA: A single file defines the secure computation. To execute, one must define cryptographic parameters, set up a certificate authority, and run scripts to compile and execute the program.

```
sum = sint(0)
for i in range(3):
  x1 = sint(i)
  x2 = sint(i*2)
  prod = x1 * x2
  sum = sum + prod
print_ln("%s", sum.reveal())
```

h) Sharemind: Examples require a SecreC file that defines the function and may include supporting files for input and output. Execution depends on which platform is used, but the provided VMs include scripts to compile and execute supporting and secure code.

```
import shared3p;
domain pd_shared3p shared3p;
void main() {
    pd_shared3p uint64 [[1]] a =
        argument("a");
    pd_shared3p uint64 [[1]] b =
        argument("b");
    pd_shared3p uint64 c = sum(a * b);
    publish("c", c);
}
```

i) PICCO: Examples require a function defined in PICCO's C extension, two configuration files, and encoded input files. Execution is a multi-step process: users must define certificates and running several custom command-line tools.

PICCO includes an inner product operator.

```
public int LEN = 10;
public int main() {
    int A[LEN], B[LEN];
    smcinput(A,1,LEN);
    smcinput(B,2,LEN);
    int p = A @ B;
    smcoutput(p,1);
    return 0;
}
```

j) Frigate: Examples are contained in a single file and compiled with a command-line tool.

```
  int result = 0;
  for(sint i=0; i<LEN; i++) {
    result = result +
        (alice.data[i] * bob.data[i]);
  }
  output1 = result;
  output2 = result;
```

k) CBMC-GC: Examples are contained in a C file and compiled and simulated with a simple Make system.

```
  int product = 0;
  for( int i=0; i<LEN; i++) {
    product += INPUT_A.xs[i] * INPUT_B.xs[i];
  }
  return product;
```