



CRYPTDOUGH: A Unified Analytics Engine for Secure Multiparty Computation

Muhammad Faisal
Boston University
mfaisal@bu.edu

Alessandra Lanz
Boston University
alanz@bu.edu

Sam Buxbaum
Boston University
sambux@bu.edu

Adam Godel
Boston University
agodel@bu.edu

Vasiliki Kalavri
Boston University
vkalavri@bu.edu

Mayank Varia
Boston University
varia@bu.edu

John Liagouris
Boston University
liagos@bu.edu

Abstract

We present CRYPTDOUGH, a unified analytics engine for secure multiparty computation (MPC). CRYPTDOUGH enables multiple distrusting parties to jointly execute a data analysis pipeline on their private inputs and learn nothing beyond the result (e.g., aggregate statistics). Unlike existing MPC solutions that support a single threat model or workload type, CRYPTDOUGH provides built-in support for cross-domain analytics (relational, time series, ML inference) under various threat models, all within the same system runtime.

CRYPTDOUGH contributes (i) a hierarchical system design that facilitates modularity and extensibility through progressive lowering of abstractions, and (ii) the concept of *virtual vectors* that enable users to write single-threaded code across all layers of the software stack, while pushing the complexity of communication, parallelization, and memory management down to the execution engine. We show that CRYPTDOUGH generalizes the functionality of state-of-the-art MPC systems and remains competitive on the analytics they support, often outperforming them by more than 2 $\times$.

CCS Concepts: • Security and privacy → Cryptography; • Information systems → Data management systems.

Keywords: secure analytics; multi-party computation

ACM Reference Format:

Muhammad Faisal, Alessandra Lanz, Sam Buxbaum, Adam Godel, Vasiliki Kalavri, Mayank Varia, and John Liagouris. 2026. CRYPTDOUGH: A Unified Analytics Engine for Secure Multiparty Computation. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26)*, September 29–October 02, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 22 pages. <https://doi.org/10.1145/3830418.3843909>



This work is licensed under a Creative Commons Attribution 4.0 International License.

SOSP '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2585-2/2026/09

<https://doi.org/10.1145/3830418.3843909>

1 Introduction

Cryptographically secure multiparty computation (MPC) [69] is based on a deceptively simple yet powerful idea: multiple parties can work together to compute the result of a function on their private data without exposing the data to each other or other external entities. The need for such collaborative analyses emerges in various scenarios, where the societal or monetary benefit is magnified if multiple, possibly distrusting, entities allow certain computations on their proprietary datasets while maintaining control over the process.

This work is motivated by three key observations about emerging MPC applications [77]. First, **threat models and security requirements vary widely across use cases**. Some applications require protection against malicious entities [29] while others assume “honest-but-curious” parties [18]. Second, **data analysis pipelines are complex and typically include diverse workloads**. The input to many ML tasks is often the result of a relational query [5], while mobile health analytics involve time series operations (windows) followed by relational aggregations [38]. Finally, **secure data workflows should be easy to develop without cryptographic expertise**. While cryptographers typically work at the level of arithmetic and boolean functionalities, analysts prefer to write programs in high-level languages that must be agnostic to the underlying secure primitives.

This paper presents CRYPTDOUGH, the first unified analytics system for MPC that meets the above requirements without compromising security or performance. CRYPTDOUGH advances the state-of-the-art in two ways: first, it provides built-in support for mixed data workflows under various threat models and, second, it prioritizes modularity and extensibility at all layers of the software stack. This way, CRYPTDOUGH can accommodate a diverse user base (data analysts, software developers, applied cryptographers), which is necessary for democratizing MPC and for keeping up with the rapidly evolving field of cryptographic computing.

1.1 Limitations of existing systems

CRYPTDOUGH addresses important limitations of prior work:

No built-in support for mixed analytics. Existing MPC systems for data analytics (§8) target a single workload

type, e.g., machine learning, relational queries, or time series computations. The notable absence of holistic approaches presents significant barriers to practical deployments, as stakeholders are typically reluctant to invest in solutions tailored to one application domain. For many real use cases that involve mixed workloads, developers are left with two options: either use specialized systems for different parts of the pipeline and do the orchestration manually, or implement the entire pipeline from scratch using a general-purpose compiler framework [60]. Both options require considerable engineering effort and cryptographic expertise.

Non-extensible system designs. The vast majority of MPC-based systems for analytics are tailored to specific protocols. This specialization is the result of tight coupling between the target setting (e.g., 3-party computation [67]) and the system implementation. For instance, protocol-specific communication patterns are often deeply embedded into the runtime, and adapting the respective systems to alternative settings requires a complete reimplementaion of core components. While there exist frameworks that support multiple threat models [13, 38, 47, 96, 102], these systems are designed primarily for analysts and lack high-level abstractions for protocol and system developers to facilitate extensibility.

Limited interoperability. One might consider addressing the above limitations by composing mixed pipelines from existing specialized systems. This approach, however, presents technical challenges related to both security and performance. Even if two systems target the same threat model, they often employ fundamentally different data encodings or cryptographic schemes [88, 97]. Securely converting between representations is non-trivial, and existing tools do not provide standardized interfaces. Security aside, orchestrating multiple systems is itself a laborious task that requires integrating different runtime environments, communication libraries, data formats, programming APIs, and (often incompatible) execution models. We argue that similar limitations of plaintext systems gave birth to unified dataflow engines in the 2010s [2, 78, 98, 100]. A decade later, the field of secure computation has reached the same inflection point.

1.2 Technical challenges and contributions

CRYPTDOUGH is a system that enables something fundamentally new: executing complex analytical pipelines on secret data owned by multiple parties. The key challenge lies in architecting a software stack that is modular, easily extensible to new threat models and workloads, and achieves competitive performance without deteriorating into a “wrapper” of external black-box libraries. CRYPTDOUGH addresses this challenge by delineating software layer boundaries, defining which primitives and operators belong to each layer, and which abstractions each layer must expose to ensure composability. While prior work has identified vector abstractions as necessary for good MPC performance [13, 67], none show

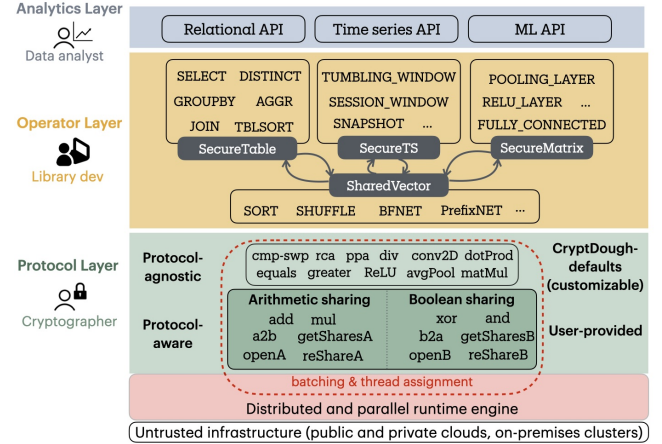


Figure 1. CRYPTDOUGH’s software stack and target users

how such abstractions can be composed to reuse low-level primitives in high-level secure operators. CRYPTDOUGH enables exactly this transparent composition and reuse.

Contributions. We make the following contributions:

- We identify a minimal set of secure MPC functionalities ($+$, $\times$, $\oplus$, $\wedge$, etc.) that are sufficient to express all workload types we consider in this work. CRYPTDOUGH exposes a vectorized interface and utilities for protocol developers to implement efficient versions of these functionalities.
- We design and implement a small set of secure, protocol-agnostic primitives (e.g., comparisons, multiplexers, etc.) that treat low-level MPC functionalities as black boxes. CRYPTDOUGH allows composing these primitives into high-level operators to ensure that user programs inherit the security guarantees of the underlying functionalities.
- We introduce *virtual vectors*, a mechanism that enables CRYPTDOUGH users to write single-threaded code across all layers of the stack and push the complexity of communication, parallelization, and memory management to the system runtime. With virtual vectors, software developers can reuse the vectorized primitives to implement secure, data-parallel operators with irregular access patterns.
- We use CRYPTDOUGH’s abstractions to implement four MPC protocols across the entire spectrum of threat models, and three secure analytics libraries (relational, time series, ML inference) that can be used to create mixed data pipelines. We show that CRYPTDOUGH generalizes the functionality of specialized systems and often exceeds their performance for the workloads they support.

The CRYPTDOUGH source code is available at <https://github.com/CASP-Systems-BU/CryptDough>.

2 CRYPTDOUGH overview

CRYPTDOUGH is an elaborate software stack in which all parts are carefully crafted to support each other, prioritizing modularity, extensibility, and separation of concerns. We

have designed CRYPTDOUGH with a hierarchy of the form: *Low-level MPC Functionalities* → *Oblivious Primitives* → *High-level Operators* → *Operator Composition*. All layers of the stack, except the bottom one, are protocol-agnostic and each layer makes black-box use of primitives from the layers below within a **fully oblivious control flow**. In practice, this means that the source code contains no conditional branches that depend (directly or indirectly) on secret data, and that user programs execute an identical sequence of operations for all inputs of the same size. Fig. 1 shows CRYPTDOUGH’s layered architecture and target users.

2.1 Design principles

CRYPTDOUGH’s architecture relies on three core principles:

1. **Configurable security.** MPC protocols have diverse trust assumptions, and no single threat model is suitable for all scenarios. For example, internal enterprise MPC deployments can assume semi-honest parties, as employees are bound by legal agreements that discourage deviations from the protocol. On the other hand, cross-organization deployments may require protection from malicious adversaries. CRYPTDOUGH accommodates this diversity by supporting MPC protocols with varying threat models.
2. **Extensibility.** The system must be easy to extend along two dimensions: (i) new MPC protocols, and (ii) new domain-specific libraries for analytics. Extensibility is achieved via CRYPTDOUGH’s layered architecture and well-defined interfaces that isolate protocol-level concerns from application logic, enabling independent evolution of both cryptographic primitives and analytics capabilities.
3. **Practical performance.** Generality must not come at the expense of performance. As we show in §7, CRYPTDOUGH is competitive with secure analytics systems that are purpose-built for specific workloads or protocols. We achieve this through a combination of efficient runtime execution strategies, optimized communication primitives, and the ability for expert users to provide custom implementations when necessary. Importantly, CRYPTDOUGH never trades security for performance.

2.2 Target users

CRYPTDOUGH’s layered architecture enables three fundamentally different types of users, each with orthogonal skills and objectives, to contribute to and benefit from the system independently, without requiring cross-domain knowledge. Specifically, CRYPTDOUGH targets the following user profiles:

A data analyst is the end-user, e.g., a data scientist, who builds data analysis pipelines. This user has domain expertise in analytics, but no cryptographic background and no prior knowledge of MPC. CRYPTDOUGH enables data analysts to compose secure dataflows using a familiar declarative DSL, similar to those offered by mainstream plaintext systems

like Apache Spark [100] and Apache Flink [24]. The DSL abstracts away all cryptographic and system-level complexity, allowing data analysts to focus on the logic of their programs.

A library developer is a domain expert in a specific analytics workload (e.g., ML, relational analytics, genomics, etc.) who wishes to extend CRYPTDOUGH with secure operators for their domain. This user knows how to design secure operators with data-independent control flows, but is not familiar with the internals of cryptographic primitives and may have limited experience with system development and optimization. CRYPTDOUGH enables library developers to implement domain-specific operators that are compatible with all supported protocols at the lowest layer.

An applied cryptographer is a user who wishes to extend CRYPTDOUGH with new MPC protocols. While possessing deep cryptographic expertise, the applied cryptographer may lack systems programming skills, parallelization knowledge, or data analytics background. CRYPTDOUGH allows such users to contribute new protocols by specifying only a minimal set of low-level functionalities, such as secure addition, multiplication, and boolean operations. To further simplify protocol implementation, CRYPTDOUGH also provides access to a communicator and a randomness generation module.

A key benefit of CRYPTDOUGH’s architecture is that *contributions from each user type immediately benefit all others* without additional integration effort. For example, new graph analytics operators contributed by a library developer would be immediately compatible with all existing MPC protocols in CRYPTDOUGH, inheriting their security guarantees and threat models. Similarly, when an applied cryptographer adds a new protocol, all existing analytics libraries can be directly used under its threat model without modification.

While CRYPTDOUGH defines clear abstractions to isolate users’ concerns, it also provides flexibility for advanced *customization* (Fig. 1, *CRYPTDOUGH-defaults*). An applied cryptographer with programming expertise can bypass default parallel implementations and provide protocol-optimized versions of higher-level functionalities for improved performance (we give an example in §4.3). CRYPTDOUGH also allows library developers to implement custom primitives at lower layers of the software stack to exploit domain-specific optimization opportunities.

2.3 Supported workloads

Real-world MPC deployments today are few and consist mainly of simple data processing, statistical analysis, secure aggregation, and basic ML [77]. CRYPTDOUGH’s main motivation is to push MPC toward more complex analytics.

To that end, CRYPTDOUGH currently supports three workload types that users can combine into end-to-end secure data pipelines: (i) relational analytics, (ii) time series computations, and (iii) ML inference. We chose these workloads specifically because they are sufficient to express real-world

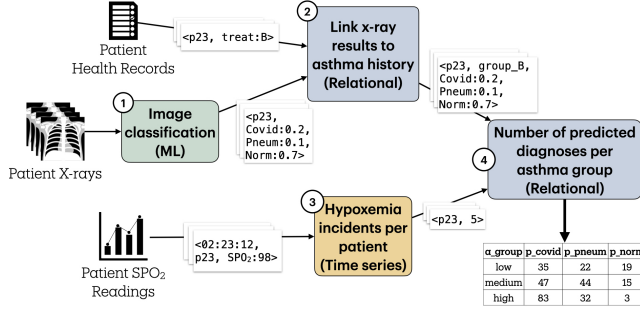


Figure 2. A mixed workflow example that analyzes diverse sensitive data from hospitals, radiology centers, and wearable devices.

use cases we have explored in collaboration with stakeholders: a mobile health application that requires secure predictions on time series data, and an education study that requires ML predictions on relational data such as exam scores, attendance records, and family histories.

Relational analytics. CRYPTDOUGH provides APIs for all common relational operators and allows their arbitrary composition into secure query plans. Specifically, it offers the same functionality as state-of-the-art relational MPC frameworks [13, 40, 84], including ORQ’s shuffling and sorting protocols and join-aggregation optimizations for efficiently evaluating acyclic multi-join queries. We have further extended this functionality with (i) fixed-point arithmetic, (ii) optimized prefix networks for efficient aggregations, and (iii) alternative sorting networks, such as bitonic sort.

Time series analytics. The time series library subsumes the functionality of prior works in this area [34, 38]. CRYPTDOUGH supports temporal operators that compute aggregates over specified time intervals, as well as tumbling and session windows (both gap-based and threshold-based). We have also implemented all time-agnostic operators from TVA and extended them to fixed-point arithmetic.

ML inference. The ML library supports common inference primitives offered by state-of-the-art systems, such as PIRanha [96] and Pigeon [47], including convolution, ReLU, pooling, and fully-connected layers. These primitives enable the construction of neural networks such as VGG-16 and AlexNet. CRYPTDOUGH also provides secure dot product and matrix multiplication primitives that can be used to implement simpler regression models.

Example workflow. Fig. 2 shows a mixed analytics workflow and Listing 1 shows how data analysts can implement it in CRYPTDOUGH. The pipeline combines image data (X-rays), tabular data (health records), and time series data (oxygen saturation measurements) to identify whether a history of hypoxemia incidents in asthmatic patients may be an indicator for diagnosing pneumonia or COVID-19 from their X-rays. The workflow begins with an image classification task on the X-rays (Step 1) to compute probabilities for three

```

1 int main(int argc, char** argv) {
2   EngineRef eng = init(argc, argv); // Engine handler
3   // Time series schema ([X] means boolean shares of X)
4   auto ts_schema = {"[TS]", "[PATIENT_ID]", "[SPO_2]",
5                     "[INCIDENT_ID]", "[INCIDENT_CNT]"};
6   // Relational schema
7   auto rel_schema = {"[PATIENT_ID]", "[ASTHMA_GROUP]"};
8   // Classification output schema
9   auto cl_schema = {"[PATIENT_ID]", "P_COVID",
10                    "P_PNEUM", "P_NORMAL"};
11   // Load secret shares
12   SecureTS<T> spo2 = eng.load(ts_schema, ...);
13   SecureTable<T> patients = eng.load(rel_schema, ...);
14   SecureMatrix<T> xrays = eng.load(IMG_SIZE, ...);
15   // Initialize model (cf. Appendix B) and load weights
16   Model vgg16 = VGG16(eng);
17   vgg16.load(...);
18   // Initialize predictions table
19   SecureTable<T> predictions = eng.init_table(cl_schema,
20                                              ROWS, ...);
21
22   // Step 1: Classify X-rays with pretrained VGG16 model
23   SecureMatrix<T> vgg_out = vgg16.forward(xrays);
24   // Extract model output and insert into predictions
25   vgg_out.as_table(predictions);
26
27   // Step 2: Join patients and prediction tables
28   auto pat_pred = patients.join(predictions,
29                                {"[PATIENT_ID]"}); // Join key
30   // Columns to keep in the join output
31   pat_pred.project(cl_schema);
32
33   // Step 3: Count hypoxemia incidents per patient
34   spo2.session_window({"[PATIENT_ID]", "[SPO_2]",
35                       "[INCIDENT_ID]", 92});
36   spo2.filter(spo2["[INCIDENT_ID]"] > 0);
37   spo2.distinct({"[PATIENT_ID]", "[INCIDENT_ID]"});
38   spo2.aggregate({"[PATIENT_ID]",
39                  {"[INCIDENT_CNT", COUNT]});
40   // Identify patients with at least one incident
41   spo2.filter(spo2["[INCIDENT_CNT]"] > 0);
42
43   // Step 4: Join outputs of steps 2 and 3 and aggregate
44   auto result = spo2.join(pat_pred,
45                           {"[PATIENT_ID]"}); // Join key
46   // Columns to keep in the join output
47   result.project({"[ASTHMA_GROUP]", "P_COVID", "P_PNEUM",
48                 "P_NORMAL"});
49   // Identify patients with high prediction probability
50   result["CNT_COVID"] = result["P_COVID"] > 0.8;
51   result["CNT_PNEUM"] = result["P_PNEUM"] > 0.8;
52   result["CNT_NORMAL"] = result["P_NORMAL"] > 0.8;
53   result.aggregate({"[ASTHMA_GROUP]", // Grouping key
54                    { // Count patients per outcome
55                      {"CNT_COVID", "CNT_COVID", COUNT},
56                      {"CNT_PNEUM", "CNT_PNEUM", COUNT},
57                      {"CNT_NORMAL", "CNT_NORMAL", COUNT},
58                    });
59   auto output = result.open_with_schema();
60 }

```

Listing 1. The workflow of Fig. 2 implemented in CRYPTDOUGH. We omit data source configuration and some minor data transformations. The program can be compiled against any of the supported MPC protocols at the lowest layer, with no modification.

outcomes: COVID-19, pneumonia, and normal. Next, the classification output is joined with a table containing asthmatic patient records and their treatment status (Step 2). The third stage computes hypoxemia incidents per patient by analyzing SPO_2 measurements using a threshold window operator (Step 3). This is a complex aggregation that first identifies time periods (windows) where a patient’s SPO_2 falls below a threshold (92) and then counts the number of

such windows. Finally, the workflow selects patients with X-ray predictions greater than 0.8 and at least one hypoxemia incident, and computes the total number of patients per outcome and asthma treatment group (Step 4).

This workflow would be extremely difficult or impossible to express in existing MPC systems. While individual steps could potentially be implemented with specialized frameworks (e.g., Piranha [96] for image classification, ORQ [13] for relational operations, TVA [38] for time series), the user would need to write custom integration code to extract output from one system and ingest it into the next. The challenge here is that different systems may target different threat models (semi-honest or malicious), making a combination impossible. But even under the same threat model, data encodings often differ. For example, Pigeon [47] and ORQ both support semi-honest, 3-party protocols but use different data sharing schemes, so passing output between them requires understanding and writing custom cryptographic code.

Alternatively, one could implement the entire workflow in a general-purpose MPC compiler, such as MP-SPDZ [60], but this would require several hundred lines of low-level code compared to the dataflow program in Listing 1. Without high-level abstractions, users must manually compose secure primitives (e.g., addition, multiplication, comparison) into secure operators. For each operator they must design an oblivious algorithm, derive padding bounds to prevent leakage, and apply batching and data parallelism to achieve reasonable performance. To get a sense of the effort required, implementing one relational query directly from low-level primitives takes ~ 800 LOC, even when using CRYPTDOUGH’s vectorized API that substantially reduces complexity. In an earlier version of our codebase that lacked vectorized abstractions, the same query required about $2\times$ more LOC.

3 Threat models and security guarantees

A typical MPC setting involves three types of entities: (i) **data holders**, who own the private input data, (ii) **computing parties**, which execute the MPC protocol on encodings of the input data to compute encodings of the result, and (iii) **output parties**, who learn the output of the computation. In CRYPTDOUGH, data holders, computing parties, and output parties are separate entities by default but may have more than one role in general; for example, data holders can also act as computing parties or analysts.

Secret sharing. CRYPTDOUGH protects data throughout the entire computation using secret sharing [88]. A data owner encodes a private value s into random ℓ -bit *shares* $s_1, s_2, \dots, s_m$ and distributes them across computing parties. No individual share reveals information about the secret, but all together are sufficient to reconstruct it. CRYPTDOUGH supports both **arithmetic** secret sharing, where shares add up to the secret using modular arithmetic (i.e., $s = s_1 + s_2 + \dots + s_m \bmod 2^\ell$), and **boolean** secret sharing, where shares

XOR to the original secret (i.e., $s = s_1 \oplus s_2 \oplus \dots \oplus s_m$). Some operations on shares are local, while others require communication between parties. For additive sharing, if parties hold shares of x and y , they can obtain shares of $x + y$ by adding their own two shares locally, and shares of $x \times y$ by executing an interactive protocol that requires exchanging $O(\ell)$ bits. For boolean secret sharing, $x \oplus y$ is local, like arithmetic addition, while $x \wedge y$ requires interaction, like arithmetic multiplication. Given protocols for secure addition and multiplication (in arithmetic sharing) or secure XOR and AND (in boolean sharing), parties can evaluate arbitrary functions on secret-shared inputs.

Security guarantees. CRYPTDOUGH is designed to provide the *full* security guarantees of the underlying MPC protocols. It protects all input, intermediate, and output data and does not leak any information to untrusted entities, not even the intermediate or output result sizes. CRYPTDOUGH guarantees data privacy even in the presence of powerful adversaries that completely control the network and can compromise t out of n computing parties ($t < n$). Adversaries can be either **semi-honest** (also known as “honest but curious”), who passively eavesdrop on the parties’ memory contents, access patterns, and network transmissions, or **malicious** (also known as “Byzantine”), who can force the parties they control to deviate from the protocol arbitrarily, e.g., by corrupting shares. We further distinguish two classes of MPC protocols: **honest-majority** protocols, where the adversary controls $t < n/2$ parties, and **dishonest-majority** protocols, where the adversary controls $t \geq n/2$ parties. In general, malicious-secure protocols for dishonest majorities are the most expensive. CRYPTDOUGH provides malicious security *with abort*, meaning that computing parties stop the computation as soon as they detect cheating to protect data privacy. We analyze CRYPTDOUGH’s security in Appendix A.

Supported protocols. CRYPTDOUGH supports protocols from the entire spectrum of threat models and can protect data against both semi-honest and malicious parties in honest and dishonest majorities. Specifically, CRYPTDOUGH supports the following state-of-the-art protocols: ABY [35] (semi-honest, dishonest majority), ABY3 [3, 74] (semi-honest, honest-majority), Fantastic Four [21, 31] (malicious, honest-majority), and SPDZ2k [30] (malicious, dishonest-majority).

4 CRYPTDOUGH’s secure abstractions

We first present CRYPTDOUGH’s oblivious data abstractions and describe the core interfaces and utilities the system exposes to users who wish to add a new MPC protocol. Then, we discuss CRYPTDOUGH’s intermediate layer of protocol-agnostic secure functionalities that can be used as black-boxes in higher-level secure operators and analytics libraries.

4.1 Oblivious data abstractions

CRYPTDOUGH offers distinct data abstractions for each of its target users (§2.2), all of which fundamentally rely on *base vector containers*. CRYPTDOUGH’s data types support a predefined set of operations on their entire content and do *not* allow access to individual elements. This vector-centric design is a core architectural choice in CRYPTDOUGH: unlike general-purpose MPC compilers [60, 101] that transform arbitrary programs into oblivious ones (e.g., by multiplexing conditionals or applying vectorization), CRYPTDOUGH operators work on vectors by construction, enabling efficient parallelization, message batching, and memory management under the covers, as we explain in §5.

Vector containers. CRYPTDOUGH defines three vector types:

A *Vector* is effectively a wrapper around `std::vector` that provides standard vectorized operations, such as element-wise addition and multiplication. It is used for local operations on the stored elements and as input to functions generating secret shares. CRYPTDOUGH also provides a *Matrix* type built on *Vector* that stores data in row-major order.

An *EVector* is a multi-vector abstraction, i.e., a container that stores a list of *Vector* objects and supports the same vectorized operations as *Vector* but applies them to all internal vectors simultaneously. For example, let *A* and *B* be two *EVector* instances, each one with two internal vectors: A_1, A_2 and B_1, B_2 respectively. Evaluating $A + B$ performs element-wise addition on each pair of the corresponding inner vectors, i.e., $A_1 + B_1$ and $A_2 + B_2$. This abstraction is essential for supporting protocols where each party maintains multiple shares per secret. For instance, replicated secret-sharing schemes [3, 31] require storing multiple shares per secret value, while malicious-secure protocols store message authentication codes (MACs) alongside shares for verifying data integrity [32]. An *EVector* is the main programming abstraction for protocol developers (cryptographers) to implement low-level MPC primitives, as we explain in §4.2.

A *SharedVector* represents an encoded view of a secret vector as seen by an untrusted party and supports secure MPC operations on its contents (secret shares). Different MPC parties have different views of the same secret vector, and these views may vary across protocols. CRYPTDOUGH supports both arithmetic and boolean secret sharing (§3) through *ASharedVector* and *BSharedVector*, which inherit from the base *SharedVector* class. This vector type serves as the core data abstraction for library developers to implement intermediate primitives and high-level operators.

Secure operations on *SharedVector* appear to users as if they were performed on local vectors. For example, given two *ASharedVectors* v_1 and v_2 of the same length, a secure element-wise multiplication is expressed as $v_1 * v_2$, where the $*$ operator triggers a secure multiplication functionality that incurs communication between parties. This complexity is handled transparently by the CRYPTDOUGH engine.

High-level containers. While cryptographers and library developers operate on vector containers, data analysts work with higher-level abstractions tailored to specific workload types. CRYPTDOUGH provides a *SecureTable* abstraction for secret-shared relational tables, a *SecureTS* type for secret-shared time series, and a *SecureMatrix* for secret-shared matrices, e.g., images. Each abstraction exposes a set of secure transformations that can be composed into declarative analytics programs and executed under MPC. For example, *SecureTable* provides relational operators such as *filter*, *join*, and *distinct*; *SecureTS* provides time-based window operators such as *tumbling_window*; and *SecureMatrix* supports ML inference operations, such as *conv*, *pooling*, and *matmul*. All these abstractions are built on top of CRYPTDOUGH’s containers; therefore, converting from one type to another is straightforward through helper functions, as shown in Listing 1.

A note on expressivity. CRYPTDOUGH’s data abstractions consciously restrict the programming model by not allowing (i) for-loops and (ii) if-conditionals on private data at the operator and analytics layer (see Fig. 1). This design facilitates optimization on vectorized operations rather than general programs, while remaining expressive enough to capture the complex analytics workflows users care about. Many plaintext frameworks, like Spark [99], Flink [24], and Google Dataflow [2], follow a similar declarative approach, where users cannot explicitly iterate over the elements of a data collection and instead apply per-element (e.g., *map*, *filter*) or per-group transformations (e.g., *reduce*, *join*). CRYPTDOUGH’s built-in operators allow the same on *SecureTable* and *SecureTS*. Experts can implement custom operators a level below, but these operators must still conform to the higher-level API. Finally, we note that not allowing conditionals on private data does not mean that CRYPTDOUGH cannot express filter predicates or conditional aggregations. Rather, data-dependent branching must be expressed with the provided oblivious operator, which evaluates both branches and assigns results using a secure multiplexer circuit.

4.2 MPC protocol abstractions

A secure MPC protocol in CRYPTDOUGH involves local computation, inter-party communication, and randomness generation. Adding a new protocol to the system requires defining the share type T , the number of parties k , and the small set of protocol-aware primitives shown inside the dark green box of Fig. 1 (we provide the full interface in Appendix E).

Functionalities are implemented as methods of an abstract protocol class that encapsulates two key modules: a *communicator* for managing inter-party message exchange, and a *randomness generator* for producing protocol-specific randomness. By inheriting from this abstract class, developers gain access to the two modules and can focus exclusively on

the protocol logic. Listing 2 shows an example implementation of the multiplication functionality for a semi-honest 2-party computation protocol (ABY) [35].

Secure multiplication in ABY proceeds in three phases (lines 12–20). First, each party invokes the randomness generator BTGen to retrieve the required number of Beaver triples

```

1 // User-defined helper function
2 Vector reconstruct(const EVector& shares) {
3     Vector remote_shares(shares(0).size()); //Receive buffer
4     // Exchange shares with other party
5     this->communicator->exchange_shares(shares(0),
6                                         remote_shares, +1);
7     return shares(0) + remote_shares; // Reconstruct
8 }
9
10 // Multiplies x, y elementwise and stores result in z
11 void multiply_a(const EVector& x, const EVector& y,
12               EVector& z) {
13     // Step 1: Pull arithmetic Beaver triples (c = a * b)
14     auto [a, b, c] = this->BTgen->getNext(x.size());
15     // Step 2: Reveal randomized inputs (interactive)
16     auto r_x = reconstruct(x + a);
17     auto r_y = reconstruct(y + b);
18     // Step 3: Compute multiplication result (local)
19     z = y*r_x - a*r_y + c; // z = x * y (noise cancels out)
20 }

```

Listing 2. Implementation of the ABY [35] secure multiplication functionality in CRYPTDOUGH. Each party executes the function on its own arithmetic shares of x and y to end up with shares of the result z .

(line 14). These are precomputed random triples of the form (a, b, c) , where $c = a \times b$, and a, b, c are secret-shared among parties using arithmetic sharing, just like inputs x and y . Next, each party randomizes x and y using a and b , respectively, and reveals the masked values to the other party (lines 16–17). This is done through a user-defined `reconstruct()` function, which internally invokes the communicator. Once the exchange completes, parties perform a final local operation that computes and unmask the shares of the result (line 19). Although the entire code appears to operate on individual shares, all steps are in fact applied to vectors using `EVector`’s overloaded operators $(+, -, *)$.

The `()` operator in lines 3 and 5 is used to access the `EVector`’s contents; that is, `shares(0)` returns the first inner vector of the shares container (at index 0). Recall from §4.1 that `EVector` is a multi-vector abstraction containing one or more inner Vectors whose number is protocol-specific. In ABY, each party receives one share of each input secret and, hence, the `EVector` type contains a single inner vector. `EVector` dimensions are set by protocol developers as a template parameter when instantiating the protocol class.

Supporting asymmetric protocols. ABY is a symmetric protocol, meaning that both parties execute the same code on different input shares. Other protocols, however, are inherently asymmetric and require parties to perform different operations based on their roles in the computation. To support such protocols, CRYPTDOUGH assigns each party a unique `partyID` in $[0, k)$, where k is the number of parties, that developers can access through the protocol class

to implement role-specific logic. For example, a developer can write `if(partyID==1) { ... }` to condition execution on a specific party. We use this technique to implement the asymmetric Fantastic Four protocol [31].

Expressing communication patterns. In line 5 of Listing 2, parties exchange messages by invoking the communicator’s `exchange_shares()` method with three arguments: a Vector of shares to send, an empty Vector to store the received shares, and a signed integer specifying the destination party. CRYPTDOUGH organizes parties in a *logical ring* topology, allowing developers to refer to other parties using relative indices: the notation $+i$ denotes the i -th successor on the ring. In our 2-party example, $+1$ indicates that each party sends to its successor and receives from its predecessor.

To support more complex communication patterns, CRYPTDOUGH provides a generalized `exchange_shares()` method that accepts four parameters: a collection of vectors to send, a collection of vectors to populate with received data, the relative indices of destination parties, and the relative indices of source parties. Protocol developers can use this method along with the `partyID` to implement arbitrary communication patterns. For example, assuming three parties on the ring, a multicast operation where party 0 sends the same vector S to parties 1 and 2 is implemented as shown below:

```

1 if (partyID==0) { // Send vector S to parties 1 and 2
2     exchange_shares({S, S}, {}, {+1,+2}, {});
3 }
4 if (partyID==1) { // Receive into vector R from party 0
5     exchange_shares({}, {R}, {}, {-1}); // +2 also works
6 }
7 if (partyID==2) { // Receive into vector R from party 0
8     exchange_shares({}, {R}, {}, {-2}); // +1 also works
9 }

```

Note that each call to `exchange_shares()` is blocking to ensure that interacting parties receive each other’s shares before proceeding. However, send and receive operations are handled by dedicated communication threads (see §5) and are performed in parallel.

Randomness generation. Many MPC protocols rely on a data-independent offline phase known as preprocessing, which generates randomness that is then used to speed up the online phase. The arithmetic triples used in Listing 2 are an example of correlated randomness. CRYPTDOUGH provides access to pseudorandom number generators (PRGs) based on AES, zero-sharing generators needed by some honest-majority protocols [3, 31], random permutations needed for 2-party oblivious shuffling [83], and Beaver triples [15, 16] (both boolean and arithmetic). Triples can be generated on-the-fly (expensive) or precomputed and stored offline, and then consumed during the online phase as needed.

Supporting other protocol types. CRYPTDOUGH’s `EVector` abstraction is designed for protocols where each input secret is encoded as one or more shares; however, some protocols

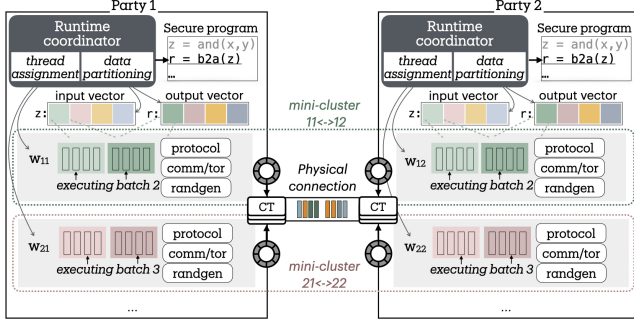


Figure 3. The CRYPTDOUGH execution model shown for 2 parties. Each party runs a CRYPTDOUGH process with a runtime coordinator that handles data partitioning and work assignment. Pairs of worker threads (w_{ij}) across parties form independent computation "mini-clusters". Message exchange is handled by communication threads (CTs) and multiplexed over physical pairwise connections.

deviate from this paradigm. A representative example is MiniMac [33], which encodes multiple input secrets into a single (array-like) shared representation. Adding such a protocol to CRYPTDOUGH requires developers to implement two additional functionalities: (i) an extraction function that defines how to retrieve a single element encoding from a packed representation, and (ii) a bit extraction function that defines how to extract individual bits needed for boolean operations.

4.3 Protocol-agnostic primitives

CRYPTDOUGH provides a set of protocol-agnostic secure primitives that use the underlying MPC functionalities as black boxes. The intermediate primitives include secure multiplexers, ripple-carry and parallel-prefix adders, comparison operators ($>$, $<$, $\geq$, $\leq$, $=$, $\neq$), compare-and-swap, ReLU, convolution, average pooling, dot product, and matrix multiplication. These primitives serve as building blocks in higher-level operators across all analytics libraries in CRYPTDOUGH. Developers can selectively override the default implementations with more efficient protocol-specific variants. For example, the default dot product operation is evaluated by invoking element-wise secure multiplication followed by aggregation. However, in replicated secret sharing protocols [3, 31], we can first execute the local part of the secure multiplication for all elements, then aggregate locally, and perform the interactive part of the multiplication in the end. This approach reduces communication complexity from linear to constant.

5 Data parallelism and virtual vectors

The vectorized abstractions described above, which require all computations to be defined as element-wise operations on vectors, provide two major benefits: (i) they allow message batching across operations to amortize MPC communication costs, and (ii) they enable efficient workload distribution by partitioning vectors across parallel workers. In this section, we first describe CRYPTDOUGH's execution model and then

explain how constructing *virtual vectors* can transparently extend parallelization to high-level user programs.

5.1 Execution model

CRYPTDOUGH's execution model is shown in Fig. 3 for an example 2-party MPC protocol. Each CRYPTDOUGH party runs on a physical machine (or VM or container) connected to other parties through a network. CRYPTDOUGH represents each party as a single OS process that includes a coordinator thread (the runtime) and a set of worker threads. Each worker thread on one party forms a *mini-cluster* with its corresponding partner threads on the other parties. Mini-clusters operate independently, maintaining separate protocol, communicator, and randomness generator objects to execute secure computations. Communication between threads within a mini-cluster is multiplexed over the physical network connection through logical channels. To avoid data copies, dedicated communication threads (CTs) manage message passing using lock-free buffers shared with mini-cluster computation threads.

All parties execute the same program as a sequence of secure primitives that operate on vectors. For each primitive, the runtime coordinator handles memory allocation, partitions the input and output vectors into disjoint subsets, and assigns each partition to a mini-cluster. Each mini-cluster further divides its partition into smaller batches. Batches within a mini-cluster are processed in order, and mini-clusters synchronize only at primitive boundaries: all mini-clusters must complete the current primitive before proceeding to the next (barrier). However, different mini-clusters may process different batches within the same primitive concurrently. This design overlaps local computation with communication across mini-clusters, improving resource utilization.

It is important to note that a secure primitive (all operations in the protocol layer of Fig. 1) represents the unit of computation assigned to a mini-cluster. A thread executes the entire control flow of a primitive on its assigned partition, potentially involving multiple communication rounds with partner threads running in remote machines, but without coordination with local worker threads. This way, high-level operators, such as sorting or aggregation, can also be transparently parallelized by the runtime, as long as they are expressed as compositions of secure vectorized primitives. We describe how CRYPTDOUGH achieves this next.

5.2 The need for vectorization

Let us first illustrate why CRYPTDOUGH enforces vectorized primitives and their composition by design. Consider the oblivious compare-and-swap operation shown in Listing 3, implemented in an imperative style, where secure operations, such as comparison ($<=$), XOR ($\wedge$), and AND ($\&$) are applied to individual elements. Function `extend_1sb()` constructs a mask of type T whose ℓ bits equal the single-bit result of the respective comparison. Note that this code snippet does

not correspond to valid CRYPTDOUGH code, as the framework does not allow iterating over individual elements of `BSharedVector` (§4).

The problem with this implementation is that some secure operations, such as comparison and AND, require communication between parties. Performing these operations

```

1 void non-vectorized-cmp-swp(BSharedVector<T>& v1,
2                             BSharedVector<T>& v2){
3     int n = v1.size(); // v1 and v2 have equal length
4     BSharedVector mask(n);
5     // Apply elementwise comparisons and compute masks
6     for (int i=0; i<n; i++) { // O(nlog|T|) comm. rounds
7         mask[i] = (v1[i] <= v2[i]).extend_lsb();
8     }
9     // Swap i-th elements obliviously if v1[i] <= v2[i]
10    BSharedVector tmp(n);
11    for (int i=0; i<n; i++) { // O(n) comm. rounds
12        tmp[i] = mask[i] & (v1[i] ^ v2[i]);
13        v1[i] ^= tmp[i];
14        v2[i] ^= tmp[i];
15    }
16 }

```

Listing 3. Baseline compare-and-swap primitive that uses non-vectorized secure operators (in orange). Each comparison (`<=`) requires $O(\log|T|)$ rounds, where $|T|$ is the length of the secret share representation in bits. For n -sized input vectors, the primitive performs $O(n \log|T|)$ communication rounds in total.

sequentially results in a large number of communication rounds, where each round acts as an expensive synchronization barrier in the distributed execution.

Although some MPC compilers can recognize such patterns [60], it is difficult to guarantee that compiler transformations will optimize arbitrary user code. In contrast, exposing a vectorized API enables CRYPTDOUGH to execute the local computation for multiple element-wise operations in parallel and exchange the resulting messages within a single network call. Listing 4 shows the same compare-and-swap program written using CRYPTDOUGH’s vectorized primitives (in blue).

```

1 void cmp-swp(BSharedVector<T>& v1, BSharedVector<T>& v2){
2     // Compare and compute mask - O(log|T|) comm. rounds
3     BSharedVector<T> mask = (v1 <= v2).extend_lsb();
4     // Swap i-th elements obliviously if v1[i] <= v2[i]
5     auto tmp = mask & (v1 ^ v2); // O(1) comm. rounds
6     v1 ^= tmp; // v1[i] = v1[i] ^ tmp[i]
7     v2 ^= tmp; // v2[i] = v2[i] ^ tmp[i]
8 }

```

Listing 4. CRYPTDOUGH’s compare-and-swap primitive. Vectorized secure operators (in blue) call into the system runtime that manages message batching under the hood. This implementation consumes the same bandwidth as Listing 3, but performs $O(\log|T|)$ communication rounds in total, independent of the input size.

5.3 Virtual vectors

While the need for vectorized operations to enable batching has been recognized by prior systems [67], the key challenge we address in CRYPTDOUGH lies in enabling *transparent reuse* of these primitives within higher-level operators. Consider the example oblivious control flow shown in Fig. 4a, which

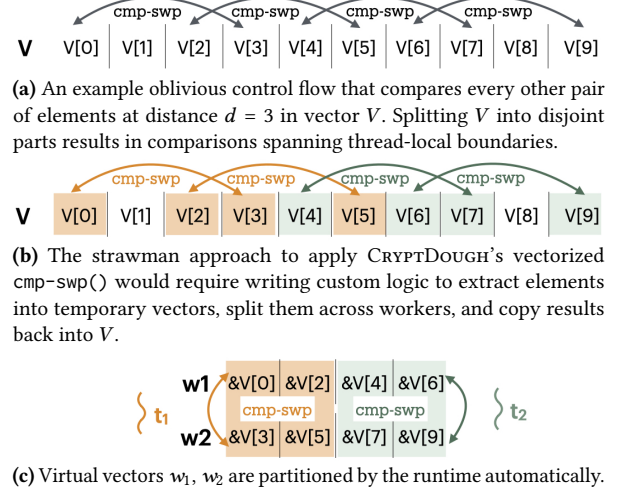


Figure 4. Virtual vectors allow transparent reuse of vectorized primitives in high-level operators and automatic parallelization.

compares and swaps every other pair of elements at distance $d = 3$ in a vector V . Such data access patterns appear frequently in oblivious prefix [49], sorting [8], and butterfly-style networks [42] that form the basis of many high-level data operators. This type of operators does not conform to a regular element-wise access pattern but instead require accessing non-contiguous elements of the same vector.

Implementing the example of Fig. 4a using CRYPTDOUGH’s vectorized API would require traversing the input vector to extract the respective elements, copying them to a new vector, and then applying the vectorized compare-and-swap primitive. Parallelizing the computation is another challenge, as simply partitioning the vector does not work. Instead, a developer would need to implement the task-to-worker assignment logic shown in Fig. 4b (see Appendix C for an example implementation). Overall, developers would need to write custom data transformation and parallelization code for every high-level operator, leading to inefficient and error-prone implementations that hinder codebase maintenance.

To address this challenge, CRYPTDOUGH provides a set of methods that allow developers to define *virtual vectors*. A virtual vector W is a restricted view of a base vector V such that $W[i] = V[f(i)]$, where f defines a mapping from indices in W to indices in V . A base vector can have multiple virtual vectors associated with it, all pointing to elements of the same underlying array (“shallow copies”). Virtual vectors are constructed via a set of functions that allow declarative expression of irregular access patterns on other vectors.

Listing 5 and Fig. 4c show the same oblivious control flow expressed using virtual vectors. Functions `offset()`, `step()`, and `limit()` define virtual vectors on V : w_1 that steps every 2 elements up to index 7, and w_2 that starts at $V[3]$ and steps every 2 elements up to V ’s size (limits are exclusive). These two vectors are passed as input to the

cmp-swp primitive, which performs the required element-wise comparisons and updates the base vector V in place.

```
1 // Both virtual vectors w1 and w2 point to elements in V
2 BSharedVector<T> w1 = V.limit(7).step(2); // Left inputs
3 BSharedVector<T> w2 = V.offset(3).step(2); // Right inputs
4 cmp-swp(w1, w2); // Updates V in place (no extra copies)
```

Listing 5. Data-parallel implementation of the oblivious operator from Fig. 4 in CRYPTDOUGH. Virtual vectors enable library developers to write single-threaded code while pushing all complexity of parallelization and memory management to the system runtime.

The CRYPTDOUGH runtime partitions and distributes virtual vectors as if they were standard (contiguous) vectors, automatically parallelizing user code while handling index translations and memory management without data copies. Thanks to the deterministic nature of oblivious programs, index translation can be computed analytically. Virtual vectors can be defined on other virtual vectors to express more complex control flows, like cyclically traversing every other element of a base vector. In Appendix D, we summarize the core data access patterns and corresponding virtual vectors that are sufficient to express all oblivious control flows in CRYPTDOUGH. We also provide their mapping functions, interfaces, and an example.

5.4 Discussion

Virtual vectors share conceptual similarities with hierarchical IRs for GPU programming [44, 80], which enable efficient thread binding without data copies or expensive synchronization. CRYPTDOUGH virtual vectors are a flat data representation tailored to the characteristics of oblivious programs, where deterministic control flow enables index computation at compile time and automatic parallelization at runtime.

This design relies on a key property of oblivious algorithms: *conflict-free data access*. One might question how CRYPTDOUGH handles potential conflicts, but in practice, oblivious control flows are structured into well-defined stages where computation within each stage is embarrassingly parallel by design. Derived from HPC and hardware literature [66], these algorithms avoid expensive synchronization within each stage, and when index overlaps occur (e.g., in odd-even merge [54]), the conflicting operations are read-only. Providing compile-time or runtime verification that developers use virtual vectors correctly is an interesting direction for future work; CRYPTDOUGH does not currently enforce such constraints.

As a final comment, virtual vectors enable automatic parallelization for a broad class of oblivious operators; however, there exist cases where we cannot easily leverage them. An example is oblivious shuffling [6], which requires random data accesses in the underlying vector. CRYPTDOUGH can still parallelize parts of the shuffling protocol using virtual vectors, but most of the work in this case runs single-threaded.

6 System implementation

CRYPTDOUGH is the result of a multi-year, collaborative development effort involving systems researchers and cryptographers. The codebase is written in C++ and uses libsodium [68] for pseudorandom number generation, libOTE [87] for MPC pre-processing, and Blaze [17] for local matrix operations. The system provides a generic, extensible communicator interface, currently supporting MPI and a socket-based implementation. CRYPTDOUGH adopts a polymorphic design that uses C++ templates to enable custom share representations and compile-time MPC protocol selection.

Implementation details. Our current implementation of the virtual vector mechanism (§5.3) precomputes and materializes index mappings to remove index translation from the critical path of the computation. We are also investigating alternative approaches, including dynamic switching between iterator-based, index precomputation, and view materialization techniques, as well as expression templates [50, 51] for efficient operator composition. CRYPTDOUGH also employs bit compression for boolean operations, which typically operate on a few bits per secret-shared element. The system provides functions to extract bits into compressed vectors and reinsert them into their original positions.

Practical considerations. CRYPTDOUGH can be deployed over LAN or WAN, either on bare-metal machines or cloud resources. Due to the communication-intensive nature of MPC, tuning parameters such as batching and the number of network threads can significantly impact performance. Our experience suggests that more batches per mini-cluster (§5.1) benefit LAN deployments by enabling computation-communication overlap, while WAN deployments benefit from fewer batches and more communication threads (CTs) to minimize expensive round trips under high RTT. By default, CRYPTDOUGH uses 12 batches with 4 CTs in LAN, and 1 batch with 16 CTs in WAN.

Scalability and fault tolerance. CRYPTDOUGH uses a single server per party and restarts the computation in case of failures. We note that real MPC deployments [77] involve very few parties (hence, the failure rate is low), and this is often mandated by the high communication cost of MPC. All state-of-the-art MPC systems focus on the outsourced setting that uses a small number of untrusted servers (in practice, machines managed by different infrastructure providers) to support any number of data owners, who outsource secret shares of their data to the servers. This is the default setting in CRYPTDOUGH and all baselines used in the paper. So far, horizontal scaling and failures have not been a main concern in the literature, since research on MPC systems is still in early stages. Adding support for fault-tolerance and distributed execution within each logical party are exciting directions for future work.

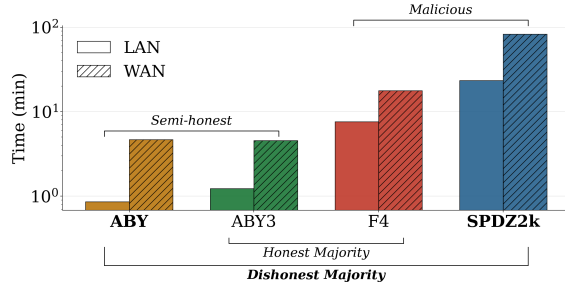


Figure 5. CRYPTDOUGH evaluation on the complex analytics pipeline of Fig. 2 across all threat models and network settings.

7 Evaluation

Our experimental evaluation consists of three parts:

- In §7.1, we evaluate CRYPTDOUGH on a real-world pipeline that includes ML, relational, and time series operations. Our results show that CRYPTDOUGH provides practical performance for multi-workload analytics across the threat model spectrum.
- In §7.2, we compare CRYPTDOUGH with five state-of-the-art MPC systems. We show that CRYPTDOUGH is competitive with specialized analytics frameworks and outperforms the popular MP-SPDZ compiler by up to 4.7×.
- In §7.3, we evaluate secure primitives and high-level operators in isolation and show that CRYPTDOUGH can effectively parallelize computations at all layers of the stack.

Experimental setting. We run experiments on four environments: (i) bm-LAN and (ii) bm-WAN use a bare-metal cluster consisting of 4 AMD EPYC 9655P 96-core machines; (iii) AWS-LAN and (iv) AWS-WAN use two g6.8xlarge AWS instances with NVIDIA L4 GPUs. LAN environments have up to 25Gbps bandwidth and 0.17–0.25ms RTT, while in the corresponding WAN environments we restrict the network characteristics to 6Gbps bandwidth and 20ms RTT. All nodes run Linux Ubuntu 24.04.4 LTS. Unless otherwise specified, we use 64-bit shares for CRYPTDOUGH and all competitors. Reported numbers are the average of three runs. For dishonest-majority protocols, we only report online times.

7.1 Performance on mixed workloads

Our first experiment evaluates CRYPTDOUGH’s capability to support end-to-end secure, mixed-analytics workflows across the entire spectrum of MPC threat models. We implement the motivating use case of Fig. 2, which consists of four stages: (i) image classification on a Kaggle dataset with 64 X-Ray images using a fine-tuned VGG16 model architecture [89], (ii) a relational join of the classification results with 1K health records of asthmatic patients, (iii) a threshold window followed by a filter and aggregation on 16K SPO_2 time series measurements, and (iv) a relational subquery consisting of a join, three filters, and a group-by with aggregation to compute the number of predicted diagnoses per asthma group. Fig. 5 shows the end-to-end execution time of the

System	MPC Protocols	Workload
ORQ [13]	ABY [35], ABY3 [3, 74]	Relational
TVA [38]	ABY3, F4 [31]	Time series
Pigeon [47]	Trio [46]	CNN inference
Piranha [96]	SecureML [75]	CNN inference
MP-SPDZ [60]	SPDZ2k [30]	CNN inference

Table 1. State-of-the-art systems along with the MPC protocols and workloads we use for comparison. MP-SPDZ is a general-purpose MPC compiler with built-in support for CNN inference.

pipeline under all MPC protocols in CRYPTDOUGH, in both bm-LAN and bm-WAN.

When configured with semi-honest security, CRYPTDOUGH evaluates the entire pipeline in $\approx 1min$ in LAN and under 5min in WAN. Malicious security is more expensive but still practical for batch analytics: CRYPTDOUGH completes execution in 18min under honest majority and just over 1h under dishonest majority in WAN. To our knowledge, this is the first demonstration of such a complex, mixed-workload analytics pipeline under MPC, and the first evaluation of secure inference using a large CNN model like VGG16 with SPDZ2k.

Although we did not attempt to implement a baseline that “stitches together” specialized systems (which may be feasible for systems with a common threat model), we can estimate the expected costs of such a solution. There would be three primary sources of overhead: (i) *initialization* time, including spawning new processes, allocating memory, and establishing network connections between parties, (ii) *secret re-sharing* time, required to convert output shares from one system into input shares compatible with the next system’s protocol, and (iii) *intermediate result materialization*, where shares must be written to persistent storage (e.g., files) by one system and subsequently loaded by the next.

CRYPTDOUGH eliminates these overheads by executing the entire workflow within a unified environment. While the next sections demonstrate that CRYPTDOUGH is competitive with state-of-the-art systems on individual workloads, the end-to-end benefits would be substantially more pronounced for mixed pipelines.

7.2 Comparison with state-of-the-art

In this section, we compare CRYPTDOUGH with five state-of-the-art systems on the workloads they support, as summarized in Table 1. We always compare systems using protocols that fall under the same threat model; however, we note that some protocols are more efficient than those in CRYPTDOUGH, which favors the baseline. For relational and time series analytics, we use ORQ [13] and TVA [38], respectively. For ML workloads, we compare with Pigeon [47] and Piranha [96], two recent systems for secure ML inference. The Pigeon paper reports that Pigeon-CPU outperforms Piranha for honest-majority MPC, which we have validated independently. Hence, we compare with Piranha only in the

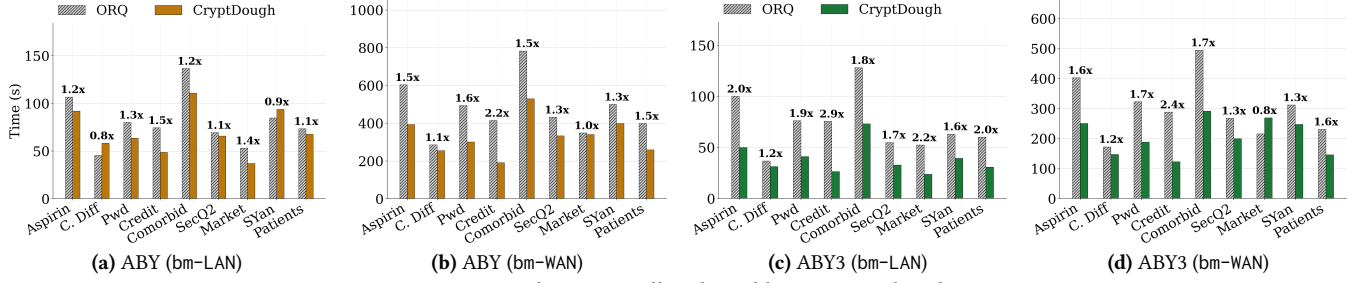


Figure 6. Comparison with ORQ on all real-world queries used in the ORQ paper

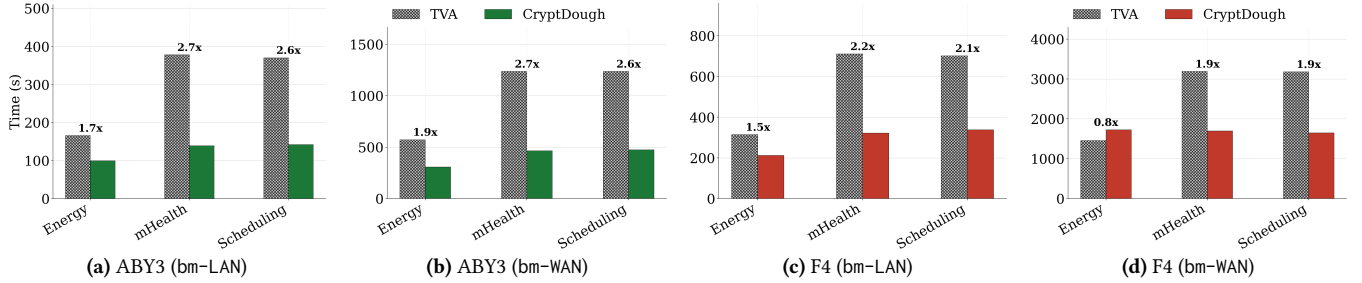


Figure 7. Comparison with TVA on all real-world applications used in the TVA paper

semi-honest dishonest-majority setting, which Pigeon does not support. For malicious security with dishonest majority, we compare with the MP-SPDZ MPC compiler [60]. While MP-SPDZ does not provide built-in support for relational or time series analytics, it offers an API for CNN inference under SPDZ2k [59] and a custom AlexNet implementation that we use for comparison.

Relational analytics. We compare with ORQ on the 9 real-world queries in the ORQ paper: *Aspirin*, *C. Diff*, *Password*, *Credit Score*, *Comorbidity*, *SecQ2*, *Market Share*, *SYan*, and *Patients*. We set the input size to scale factor 1 (≈ 5 M rows per table), as in the ORQ paper, and use oblivious quicksort for both systems. We run experiments with two of the three protocols ORQ supports, since a recent paper [21] identified an attack on the Fantastic Four (F4) protocol, and ORQ’s security fix was in progress at the time of writing [25] (to the best of our understanding, CRYPTDOUGH’s implementation of F4 is not vulnerable to this attack)¹. We use ORQ’s default configuration for both systems: 16 threads with 4 connections in LAN and 16 connections in WAN. Fig. 6 shows that CRYPTDOUGH achieves an average speedup of 1.5 $\times$ over ORQ.

Time series analytics. For the comparison with TVA, we use the three real-world applications in the TVA paper [38, §6.2]: *monitoring energy consumption*, *mobile health analytics*, and *job scheduling optimization*. We use 2^{22} rows per time series, which is the largest input TVA reports results for. For a fair comparison, we configure CRYPTDOUGH to use bitonic sort and 32-bit shares (with 64-bit timestamps), like TVA, and set the number of threads to 16 for both systems (TVA’s default). We show results for both TVA protocols,

¹ORQ has since fixed its Fantastic Four protocol implementation [12, 14].

ABY3, and F4 (as far as we know, TVA’s F4 implementation is not vulnerable to the attack [21]). The results in Fig. 7 demonstrate that CRYPTDOUGH outperforms TVA in all but one configuration, offering up to 2.7 $\times$ lower latency.

ML inference: CRYPTDOUGH vs. Pigeon. For the comparison with Pigeon, we evaluate inference using (i) AlexNet on CIFAR-10, and (ii) VGG16 on CIFAR-10 and ImageNet. We use 192 images (the largest batch size reported in the Pigeon paper) and we configure both systems with 24 processes. Pigeon is a highly-optimized secure ML system that can leverage AVX instructions and a communication-efficient asymmetric 3-party protocol [46]. It further provides implementations of custom circuits for boolean addition to accelerate ReLU layers. We enable all of these optimizations in our experiments.

Table 2 shows the results. Despite being a general-purpose framework without any ML-specific optimizations or protocols, CRYPTDOUGH is competitive with Pigeon and even outperforms it in LAN. In a high-RTT setting, Pigeon’s custom ReLU gives it an advantage, as it significantly reduces the amount of communication required.

ML inference: CRYPTDOUGH vs. Piranha. Piranha is a GPU-only inference framework. Since our bare-metal cluster does not have GPUs, we run Piranha in AWS-LAN and AWS-WAN on instances having the same GPU type as the one used for Piranha in the Pigeon paper [47]. We run the corresponding CRYPTDOUGH experiments in bm-LAN and bm-WAN. Since Piranha cannot leverage CPU multi-threading, we configure CRYPTDOUGH to only use one thread. We also use the same models and datasets as in the previous experiment, but we restrict the input to 8 images for VGG16 on ImageNet

		AlexNet (CIFAR-10)	VGG16 (CIFAR-10)	VGG16 (ImageNet)
LAN	Pigeon	0.49 s	1.13 s	52.3 s
	CRYPTDOUGH	0.17 s	1.11 s	51.0 s
	Speedup	2.9×	1.0×	1.0×
WAN	Pigeon	2.68 s	5.52 s	183.5 s
	CRYPTDOUGH	3.92 s	15.3 s	219.6 s
	Speedup	0.7×	0.4×	0.8×

Table 2. Comparison with Pigeon (Trio vs. ABY3)

		AlexNet (CIFAR-10)	VGG16 (CIFAR-10)	VGG16 (ImageNet)
LAN	Piranha	0.57 s	31.6 s	47.3 s
	CRYPTDOUGH	0.46 s	14.2 s	30.1 s
	Speedup	1.2×	2.2×	1.6×
WAN	Piranha	3.58 s	80.2 s	164.4 s
	CRYPTDOUGH	9.20 s	56.2 s	92.3 s
	Speedup	0.4×	1.4×	1.8×

Table 3. Comparison with Piranha (SecureML vs. ABY)

	MP-SPDZ	CRYPTDOUGH	Speedup
LAN	28.45 s	24.07 s	1.2×
WAN	581.9 s	124.7 s	4.7×

Table 4. Comparison with MP-SPDZ (SPDZ2k) on AlexNet [58]

because Piranha runs out of memory with a larger input size. Table 3 shows that, even with a single thread, CRYPTDOUGH outperforms Piranha, in all but one case, achieving up to 2.2× faster inference.

ML inference: CRYPTDOUGH vs. MP-SPDZ. Table 4 shows the comparison with MP-SPDZ. For this experiment, we run the custom AlexNet architecture provided in the public MP-SPDZ repository [58], and we implement the same architecture in CRYPTDOUGH. We configure both systems with 16 threads, no preprocessing, and an input size of 192 images. CRYPTDOUGH outperforms MP-SPDZ in both LAN and WAN; for WAN it achieves 4.7× lower inference time, demonstrating state-of-the-art performance also under the strongest MPC threat model, which is not supported by any of the above systems.

7.3 Evaluating CRYPTDOUGH’s parallelization

Next, we assess how well the CRYPTDOUGH runtime can automatically parallelize low-level MPC functionalities and high-level secure operators. To this end, we evaluate four secure primitives that serve as building blocks for CRYPTDOUGH’s analytics libraries: comparison, a Ripple-Carry Adder (RCA), a Parallel-Prefix Adder (PPA), and convolution. We use vectors with 2^{28} elements for comparison and adders, and 256 instances of the largest VGG16-ImageNet Conv2D layer for convolution. Fig. 8a, 8b, and 8c show the execution time as we increase the number of threads from 1 to 32, for ABY3

in bm-LAN. Experiments with other CRYPTDOUGH protocols are given in Appendix F.

Comparison and addition scale almost perfectly with up to 8 threads, achieving $\approx 2\times$ speedup at every step. Increasing the number of threads further amplifies the overheads of runtime coordination and communication, though, CRYPTDOUGH still achieves $\approx 1.5\times$ speedup going from 16 to 32 threads. Convolution is more challenging to scale, as the ratio of communication to computation is high, and overlapping provides only modest benefits: CRYPTDOUGH achieves between 1.45× and 1.1× speedup when doubling the number of threads.

To understand the impact of primitive composition on scaling, we run bitonic sort and quicksort on a vector of 2^{26} elements. These are two high-level oblivious operators that make black-box use of lower-level functionalities and form the basis for many of CRYPTDOUGH’s relational and time series operations. The results in Fig. 8d show that the CRYPTDOUGH runtime can effectively parallelize both, even though the library developer expressing the oblivious control flow of these sorting networks writes single-threaded code.

8 Related work

MPC systems for analytics. In the ML space, existing MPC-based systems support a range of tasks, including linear model fitting [41, 75, 82, 102, 103], neural network inference [55, 73, 81], and even training [7, 47, 53, 63, 71, 90, 92, 96]. For relational analytics, systems like ORQ [13], Conclave [91], and others [9–11, 28, 40, 45, 67, 84, 95, 104] support a broad class of database queries on tabular data. The time series and graph analytics space are comparatively less explored; Waldo [34] and TVA [38] support time series and some basic relational operations (filters and aggregations), while GraphSC [79] and others [4, 64] target graph analysis that CRYPTDOUGH does not currently support.

Unlike CRYPTDOUGH, all these systems are designed for a single workload type—ML, relational, time series, or graph analytics—and cannot naturally support mixed pipelines that arise in practice. Moreover, the vast majority of these systems is tailored to a single threat model, further limiting their applicability. CRYPTDOUGH is the first MPC-based analytics system to report results across the full threat model spectrum, encompassing both semi-honest and malicious adversaries in honest- and dishonest-majority settings.

MPC compilers. Frameworks like MP-SPDZ [60], MOTION [20], and others [1, 26, 27, 48, 65, 70, 72, 85, 94, 101] have made MPC more accessible by allowing users to write arbitrary programs in high-level procedural languages (e.g., subsets of Python or C) and compile them into secure MPC circuits. This line of work focuses on cost-based compiler optimizations, such as automatic vectorization, expression rewriting [62], or mixing different MPC protocols in the same user program [20, 39]. While CRYPTDOUGH could incorporate

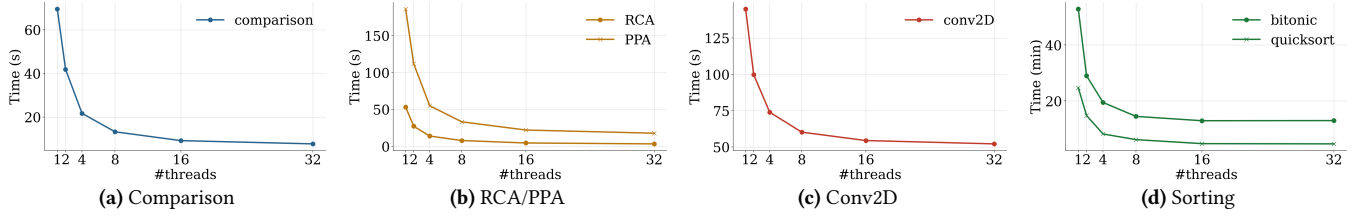


Figure 8. CRYPTDOUGH's performance as we increase the number of worker threads per party (ABY3, bm-LAN)

such techniques to optimize UDFs, our approach is fundamentally different. CRYPTDOUGH does not allow users to write arbitrary programs; in contrast, it provides vectorized, data-parallel operators for analytics that can be composed arbitrarily to create mixed workflows like the one in Fig. 2. Expressing such pipelines in the above frameworks is technically possible, but would have the same hurdles we discussed in §2.3. This holds even for systems like MP-SPDZ and MOTION that provide built-in support for CNN inference, since users would still need to implement the relational and time series operators directly from low-level primitives.

Custom MPC solutions. There are also several MPC solutions that target specific use cases and workloads, including aggregate statistics [19, 23, 52], private matching [22, 76], and heavy hitters [29, 86], among others. Splinter [93] uses function secret sharing to evaluate private queries on public data, while Flock [56] targets secure key management.

9 Conclusion

We presented CRYPTDOUGH, the first unified analytics engine for secure multiparty computation. CRYPTDOUGH provides built-in support for relational queries, time series computations, ML inference tasks, and their composition, generalizing the functionality of prior solutions, without compromising security or performance. We believe CRYPTDOUGH will be a useful tool towards pushing MPC beyond current niche applications.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their thoughtful comments that substantially improved the paper, and the artifact evaluators for reproducing the results of this work. We also thank our collaborators Nicole Spartano, Lisa Quintiliani, Edwin Boudreaux, Ola Ozernov-Palchik, Rhoda Au, the Boston Women's Workforce Council (BWFC), the Massachusetts Technology Collaborative (MassTech), the Massachusetts Executive Office of Labor and Workforce Development (EOLWD), Bosch GmbH, and Red Hat Inc for providing real use cases that have influenced CRYPTDOUGH's design. We are also grateful to the following individuals for their useful feedback on system design and their code reviews: Professors Jonathan Appavoo and Azer Bestavros; and Amer Elsheikh, Eli Baum, Eric Chen, Ian

Saucy, Iman Attia, Islam Faisal, Jerry Zhang, Mohamed Ayman, Mohamed Oscar, Nitin Mathai, Selene Wu, and Thomas Unger. This paper is based upon work supported by NSF awards No. 2209194, 2541869, and 2613424; by REU supplement awards No. 2432612 and 2326580; by an Amazon Research Award; and by a Foresight Institute AI for Science & Safety grant. Initial development and prototyping was done with computing resources provided by the Chameleon [57] and CloudLab [36] research testbeds supported by the National Science Foundation.

References

- [1] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 2021. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 740–755. doi:10.1145/3453483.3454074
- [2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1792–1803. doi:10.14778/2824032.2824076
- [3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 805–817.
- [4] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, Republic of Korea) (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 610–629. doi:10.1145/3460120.3484560
- [5] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. 2018. From Keys to Databases - Real-World Applications of Secure Multi-Party Computation. *Comput. J.* 61, 12 (2018), 1749–1771.
- [6] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (Los Angeles, CA, USA) (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 125–138. doi:10.1145/3548606.3560691
- [7] Nuttapon Attrapadung, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Takahiro Matsuda, Ibuki Mishina, Hiraku Morita, and Jacob C. N. Schuldt. 2021. Adam in Private: Secure and Fast Training of Deep

- Neural Networks with Adaptive Moment Estimation. <https://eprint.iacr.org/2021/736>
- [8] Kenneth E. Batchier. 1968. Sorting Networks and Their Applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*. 307–314. doi:10.1145/1468075.1468121
 - [9] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *Proc. VLDB Endow.* 10, 6 (2017), 673–684. doi:10.14778/3055330.3055334
 - [10] Johes Bater, Xi He, William Ehrlich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018), 307–320.
 - [11] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.
 - [12] Eli Baum. 2026. To Catch a Cheater: Some notes on the malicious (in)security of ORQ. *IACR Cryptol. ePrint Arch.* 2026 (2026), 1152.
 - [13] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. 2025. ORQ: Complex Analytics on Private Data with Strong Security Guarantees. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 802–833.
 - [14] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. 2026. ORQ: Scaling Complex Multiparty Computations to Large Private Datasets. *ACM Trans. Comput. Syst.* (July 2026). doi:10.1145/3820248 Just Accepted.
 - [15] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *Proceedings of the 11th Annual International Cryptology Conference (CRYPTO)* (Santa Barbara, California, USA). 420–432. doi:10.1007/3-540-46766-1_34
 - [16] Donald Beaver. 1995. Precomputing Oblivious Transfer. In *Proceedings of the 15th Annual International Cryptology Conference (CRYPTO)* (Santa Barbara, California, USA). 97–109. doi:10.1007/3-540-44750-4_8
 - [17] Blaze community. 2020. Blaze C++ Math Library. <https://bitbucket.org/blaze-lib/blaze/src/master/>. [Online; accessed March 2026].
 - [18] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. 2016. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *Proceedings on Privacy Enhancing Technologies (PoPETS)* 2016, 3 (2016), 117–135. <http://www.degruyter.com/view/j/popets.2016.2016.issue-3/popets-2015-0019/popets-2016-0019.xml>
 - [19] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. [n.d.]. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM Conference on Computer and Communications Security*. ACM, 1175–1191.
 - [20] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. MOTION – A Framework for Mixed-Protocol Multiparty Computation. *ACM Trans. Priv. Secur.* 25, 2, Article 8 (March 2022), 35 pages. doi:10.1145/3490390
 - [21] Andreas Brüggemann and Thomas Schneider. 2026. When Trying to Catch Cheaters Breaks the MPC: Breaking and Fixing Delayed Consistency Checks in Trident, Fantastic Four, SWIFT, and Quad (Full Version). *Cryptology ePrint Archive*, Paper 2026/234. <https://eprint.iacr.org/2026/234>
 - [22] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlasin. 2020. Private Matching for Compute. *Cryptology ePrint Archive*, Report 2020/599. <https://eprint.iacr.org/2020/599>.
 - [23] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. 2010. SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security)* (Washington, DC, USA). 15–15. <http://dl.acm.org/citation.cfm?id=1929820.1929840>
 - [24] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).
 - [25] CASP-Systems. 2025. ORQ github repository: 4pc security notice. <https://github.com/CASP-Systems-BU/orq/commit/a4e4f8270fb7af5f7c8f396240e192b03586ca67>.
 - [26] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 496–511. doi:10.1109/EuroSP.2019.00043
 - [27] E. Chen, J. Zhu, A. Ozdemir, R. S. Wahby, F. Brown, and W. Zheng. 2023. Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 848–863. doi:10.1109/SP46215.2023.10179397
 - [28] Sherman S. M. Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. 2009. Two-Party Computation Model for Privacy-Preserving Queries over Distributed Databases. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society. <https://www.ndss-symposium.org/ndss2009/two-party-computation-model-privacy-preserving-queries-over-distributed-databases/>
 - [29] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) (*NSDI'17*). USENIX Association, USA, 259–282.
 - [30] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPDZ_{zk}: Efficient MPC mod 2^k for Dishonest Majority. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 769–798.
 - [31] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic four: {Honest-Majority} {Four-Party} secure computation with malicious security. In *30th USENIX Security Symposium (USENIX Security 21)*. 2183–2200.
 - [32] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. 2013. Practical covertly secure MPC for dishonest majority—or: breaking the SPDZ limits. In *European Symposium on Research in Computer Security*. Springer, 1–18.
 - [33] Ivan Damgård and Sarah Zakarias. 2013. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography Conference*. Springer, 621–641.
 - [34] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A Private Time-Series Database from Function Secret Sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2450–2468. doi:10.1109/SP46214.2022.9833611
 - [35] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. <https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation>
 - [36] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet,

- Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. <https://www.usenix.org/conference/atc19/presentation/duplyakin>
- [37] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO (2) (Lecture Notes in Computer Science, Vol. 12171)*. Springer, 823–852.
- [38] Muhammad Faisal, Jerry Zhang, John Liagouris, Vasiliki Kalavri, and Mayank Varia. 2023. TVA: A multi-party computation system for secure and expressive time series analytics. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5395–5412. <https://www.usenix.org/conference/usenixsecurity23/presentation/faisal>
- [39] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. 2022. CostCO: An automatic cost modeling framework for secure multi-party computation. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 140–153. doi:10.1109/EuroSP53844.2022.00017
- [40] Wenjing Fang, Shunde Cao, Guojin Hua, Junming Ma, Yongqiang Yu, Qunshan Huang, Jun Feng, Jin Tan, Xiaopeng Zan, Pu Duan, et al. 2024. SecretFlow-SCQL: A Secure Collaborative Query Platform. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3987–4000.
- [41] Adrià Gascón, Philipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. 2017. Privacy Preserving Distributed Linear Regression on High-Dimensional Data. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*. 345–364.
- [42] Michael T. Goodrich. 2011. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*. ACM, 379–388.
- [43] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. 2016. MPC-Friendly Symmetric Key Primitives. In *CCS*. ACM, 430–443.
- [44] Bastian Hagedorn, Ben Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 302–313. doi:10.1145/3582016.3582018
- [45] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable Collaborative Analytics System on Private Database with Malicious Security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1740–1753. doi:10.1109/ICDE53745.2022.00176
- [46] Christopher Harth-Kitzerow, Ajith Suresh, Yongqin Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. 2025. High-Throughput Secure Multiparty Computation with an Honest Majority in Various Network Settings. *Proceedings on Privacy Enhancing Technologies* 1 (2025), 250–272.
- [47] Christopher Harth-Kitzerow, Yongqin Wang, Rachit Rajat, Georg Carle, and Murali Annavaram. 2025. PIGEON: A high throughput framework for private inference of neural networks using secure multiparty computation. *Proceedings on Privacy Enhancing Technologies* (2025).
- [48] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewicz. 2019. SoK: General Purpose Compilers for Secure Multiparty Computation. In *IEEE Symposium on Security and Privacy*. IEEE, 1220–1237.
- [49] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.
- [50] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing* 34, 2 (2012), C42–C69.
- [51] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. High performance smart expression template math libraries. In *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 367–373.
- [52] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. 2020. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *EuroS&P*. IEEE, 370–389.
- [53] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2024. Orca: FSS-based Secure Training and Inference with GPUs. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 597–616. doi:10.1109/SP54263.2024.00063
- [54] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. 2011. Secure Multi-Party Sorting and Applications. *IACR Cryptol. ePrint Arch.* 2011 (2011), 122. <http://eprint.iacr.org/2011/122>
- [55] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1651–1669. <https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar>
- [56] Darya Kaviani, Sijun Tan, Pravein Govindan Kannan, and Raluca Ada Popa. 2024. Flock: A Framework for Deploying On-Demand Distributed Trust. In *USENIX Symposium on Operating Systems Design and Implementation*.
- [57] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Collieran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 219–233. <https://www.usenix.org/conference/atc20/presentation/keahey>
- [58] Marcel Keller. 2016. MP-SPDZ AlexNet. https://github.com/data61/MP-SPDZ/blob/master/Scripts/torch_cifar_alex_import.py. [Online; accessed March 2026].
- [59] Marcel Keller. 2016. MP-SPDZ Machine Learning. <https://mp-spdz.readthedocs.io/en/latest/machine-learning.html>. [Online; accessed March 2026].
- [60] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.
- [61] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. 2017. Faster Secure Multiparty Computation of AES and DES Using Lookup Tables. In *ACNS (LNCS, Vol. 10355)*. Springer, 229–249.
- [62] Florian Kerschbaum. 2013. Expression Rewriting for Optimizing Secure Computation. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)* (San Antonio, Texas, USA), 49–58. doi:10.1145/2435349.2435356
- [63] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. 2020. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*.
- [64] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) (CCS '24). Association for Computing Machinery, New York, NY, USA, 4017–4031. doi:10.1145/3658644.3670393
- [65] KU Leuven. [n. d.]. SCALE-MAMBA Software. <https://homes.esat.kuleuven.be/~nsmart/SCALE/>. [Online; accessed April 2026].
- [66] H.T. Kung. 1980. The Structure of Parallel Algorithms. *Advances in Computers*, Vol. 19. Elsevier, 65–112. doi:10.1016/S0065-2458(08)

- 60033–9
- [67] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. Secrecy: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation*.
 - [68] The libsodium community. 2026. libsodium: A modern, portable, easy to use crypto library. <https://libsodium.org/>. [Online; accessed March 2026].
 - [69] Yehuda Lindell. 2020. Secure multiparty computation. *Commun. ACM* 64, 1 (dec 2020), 86–96. doi:10.1145/3387108
 - [70] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. OblivM: A Programming Framework for Secure Computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)* (San Jose, California, USA). 359–376. doi:10.1109/SP.2015.29
 - [71] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. 2023. SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 17–33. <https://www.usenix.org/conference/atc23/presentation/ma>
 - [72] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay-Secure Two-Party Computation System.. In *USENIX Security Symposium*, Vol. 4. San Diego, CA, USA.
 - [73] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium, USENIX Security 2020, August 12–14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2505–2522. <https://www.usenix.org/conference/usenixsecurity20/presentation/mishra>
 - [74] Payman Mohassel and Peter Rindal. 2018. ABY³: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 35–52. doi:10.1145/3243734.3243760
 - [75] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.
 - [76] Dimitris Mouris, Daniel Masny, Ni Trieu, Shubho Sengupta, Prasad Buddhavarapu, and Benjamin M. Case. 2024. Delegated Private Matching For Compute. *Proc. Priv. Enhancing Technol.* 2024, 2 (2024), 49–72.
 - [77] MPC Deployments. 2024. MPC Deployments: A hub for real-world MPC deployments. <https://mpc.cs.berkeley.edu>.
 - [78] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (Farmington, Pennsylvania) (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. doi:10.1145/2517349.2522738
 - [79] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 377–394. doi:10.1109/SP.2015.30
 - [80] NVIDIA. 2025. Tile IR. <https://docs.nvidia.com/cuda/tile-ir/latest/>. [Online; accessed March 2026].
 - [81] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. 2024. BOLT: Privacy-Preserving, Accurate and Efficient Inference for Transformers. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19–23, 2024*. IEEE, 4753–4771. doi:10.1109/SP54263.2024.00130
 - [82] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23–26, 2020*. The Internet Society. <https://www.ndss-symposium.org/ndss-paper/blaze-blazing-fast-privacy-preserving-machine-learning/>
 - [83] Stanislav Peceny, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. 2024. Efficient Permutation Correlations and Batched Random Access for Two-Party Computation. *Cryptology ePrint Archive, Paper 2024/547*. <https://eprint.iacr.org/2024/547>
 - [84] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. <https://www.usenix.org/conference/usenixsecurity21/presentation/poddar>
 - [85] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. Washington, DC, USA, 655–670. doi:10.1109/SP.2014.48
 - [86] Mayank Rathee, Yuwen Zhang, Henry Corrigan-Gibbs, and Raluca Ada Popa. 2024. Private Analytics via Streaming, Sketching, and Silently Verifiable Proofs. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19–23, 2024*. IEEE, 3072–3090. doi:10.1109/SP54263.2024.00245
 - [87] Peter Rindal and Lance Roy. 2016. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. <https://github.com/osu-crypto/libOTe>. [Online; accessed April 2026].
 - [88] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. doi:10.1145/359168.359176
 - [89] Daniel Stankowski. 2023. Multiclass image classification (X-ray). https://github.com/danielstankw/VGG16_Xray_Image_Classifier. [Online; accessed March 2026].
 - [90] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *SP. IEEE*, 1021–1038.
 - [91] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25–28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 3:1–3:18. doi:10.1145/3302424.3303982
 - [92] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proceedings on Privacy Enhancing Technologies*.
 - [93] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical Private Queries on Public Data. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (Boston, MA, USA) (NSDI'17)*. USENIX Association, USA, 299–313.
 - [94] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. <https://github.com/emp-toolkit>.
 - [95] Yilei Wang and Ke Yi. 2021. *Secure Yannakakis: Join-Aggregate Queries over Private Data*. Association for Computing Machinery, New York, NY, USA, 1969–1981. <https://doi.org/10.1145/3448016.3452808>
 - [96] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A {GPU} platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*. 827–844.
 - [97] Andrew C. Yao. 1982. Protocols for Secure Computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (AFCS)*. Washington, DC, USA, 160–164. doi:10.1109/SFCS.1982.88
 - [98] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (San Diego, California)*

- (OSDI'08). USENIX Association, USA, 1–14.
- [99] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*. 15–28.
- [100] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. doi:10.1145/2934664
- [101] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: a general-purpose compiler for private distributed computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (Berlin, Germany) (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 813–826. doi:10.1145/2508859.2516752
- [102] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. 2021. Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2723–2740. <https://www.usenix.org/conference/usenixsecurity21/presentation/zheng>
- [103] Wenting Zheng, Raluca A. Popa, Joseph Gonzalez, and Ion Stoica. 2019. Helen: Maliciously Secure Cooperative Learning for Linear Models. *2019 IEEE Symposium on Security and Privacy (SP)* (2019), 724–738.
- [104] Peizhao Zhou, Xiaojie Guo, Pinzhi Chen, Tong Li, Siyi Lv, and Zheli Liu. 2024. Shortcut: Making MPC-based Collaborative Analytics Efficient on Dynamic Databases. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (Salt Lake City, UT, USA) (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 854–868. doi:10.1145/3658644.3690314

Appendix

We provide a security analysis in §A, an example VGG16 implementation in §B, and a custom parallelization example in §C. We then summarize CRYPTDOUGH’s core access patterns and protocol interface in §D and §E, respectively. In §F, we provide additional results on scalability and throughput. Appendices are not included in the peer-review process.

A Security Analysis

A.1 Security of composition

CRYPTDOUGH uses the functionalities of its underlying MPC protocols in a black-box manner, supporting mixed-mode operations and conversions. CRYPTDOUGH is designed so that each layer directly inherits the security guarantees of the layer below it, and ultimately of the underlying MPC protocol. Concretely, our higher-level operators are built from the secure primitives exposed by the protocol layer (e.g., addition, multiplication, XOR-gate, and AND-gate) and listed in Table 5. Higher-level operators invoke these primitives without inspecting or branching on any secret value. Their control flow is data-independent, so an operator’s execution and access pattern reveal nothing about its inputs, outputs, or intermediate data. Primitives are instantiated following the constructions of Demmler et al. [35], Araki et al. [3, 74], Dalskov et al. [21, 31], and Cramer et al. [30].

Thanks to this black-box model and data-independent composition, the security of CRYPTDOUGH’s operators and analytics layers follows in the standard arithmetic black-box (ABB) model: any protocol that securely realizes the primitive functionalities can be composed, and the resulting higher-level protocol is secure by the standard composition argument. It thereby inherits the privacy and correctness guarantees—as well as the threat model (semi-honest or malicious)—of the instantiated primitives. Prior work [37, 38, 43, 61] has used a similar argument to prove system security.

A.2 Security assumptions

Like all MPC systems for analytics, CRYPTDOUGH has the entire codebase in the trusted computing base (TCB); that is, a flaw in any part of the implementation can foil the MPC security guarantees. However, CRYPTDOUGH does not rely on trusted compute or storage; in other words, the security guarantees hold even if an administrator observes the state of a server during and between computations. As a result, CRYPTDOUGH can be deployed in a typical outsourced setting where data owners secret-share their data to untrusted computing parties. The network layer is also considered untrusted, so CRYPTDOUGH needs to establish authenticated, encrypted channels between servers using TLS.

CRYPTDOUGH’s fully oblivious implementation across all layers of the stack eliminates known timing-based side channels. We make no explicit attempt to prevent other forms

of side channels (e.g., cache or power). Because MPC programs are oblivious, side-channel leakage at a small number of parties is naturally mitigated: the computation behaves analogously to a masking countermeasure, revealing nothing about secret values from any single party’s execution. Formal verification of the CRYPTDOUGH software is an exciting direction for future work; the current modular architecture would greatly facilitate this process.

B Example CNN model implementation

Listing 6 shows the implementation of the fine-tuned VGG16 model [89] that we use for X-ray classification in the multi-workload pipeline of Fig. 2 and the experiments of §7.1.

```

1 Model<T, SecureMatrix, Engine> VGG16(Engine& engine) {
2     const size_t precision = 16; // Fixed-point precision
3     // Model initialization
4     Model<T, SecureMatrix, Engine> m(engine, precision);
5
6     /** Construct VGG16 model layers */
7
8     // HW(X,Y) defines layer dimensions (height,width)
9     m.conv2D(HW(224,224), 3, 64, HW(3,3), HW(1,1), HW(1,1));
10    m.relu(3211264);
11    m.conv2D(HW(224,224), 64, 64, HW(3,3), HW(1,1), HW(1,1));
12    m.avgPool(HW(224,224), 64, HW(2,2), HW(2,2), HW(0,0));
13    m.relu(802816);
14    m.conv2D(HW(112,112), 64, 128, HW(3,3), HW(1,1), HW(1,1));
15    m.relu(1605632);
16    m.conv2D(HW(112,112), 128, 128, HW(3,3), HW(1,1), HW(1,1));
17    m.avgPool(HW(112,112), 128, HW(2,2), HW(2,2), HW(0,0));
18    m.relu(401408);
19    m.conv2D(HW(56,56), 128, 256, HW(3,3), HW(1,1), HW(1,1));
20    m.relu(802816);
21    m.conv2D(HW(56,56), 256, 256, HW(3,3), HW(1,1), HW(1,1));
22    m.relu(802816);
23    m.conv2D(HW(56,56), 256, 256, HW(3,3), HW(1,1), HW(1,1));
24    m.avgPool(HW(56,56), 256, HW(2,2), HW(2,2), HW(0,0));
25    m.relu(200704);
26    m.conv2D(HW(28,28), 256, 512, HW(3,3), HW(1,1), HW(1,1));
27    m.relu(401408);
28    m.conv2D(HW(28,28), 512, 512, HW(3,3), HW(1,1), HW(1,1));
29    m.relu(401408);
30    m.conv2D(HW(28,28), 512, 512, HW(3,3), HW(1,1), HW(1,1));
31    m.avgPool(HW(28,28), 512, HW(2,2), HW(2,2), HW(0,0));
32    m.relu(100352);
33    m.conv2D(HW(14,14), 512, 512, HW(3,3), HW(1,1), HW(1,1));
34    m.relu(100352);
35    m.conv2D(HW(14,14), 512, 512, HW(3,3), HW(1,1), HW(1,1));
36    m.relu(100352);
37    m.conv2D(HW(14,14), 512, 512, HW(3,3), HW(1,1), HW(1,1));
38    m.avgPool(HW(14,14), 512, HW(2,2), HW(2,2), HW(0,0));
39    m.relu(25088);
40
41    // Task-specific layers for Xray classification (§7.1)
42    m.avgPool(HW(7,7), 512, HW(2,2), HW(2,2), HW(0,0));
43    m.relu(512);
44    m.fullyConnected(512, 256);
45    m.relu(256);
46    m.fullyConnected(256, 256);
47    m.relu(256);
48    m.fullyConnected(256, 3);
49    m.relu(3);
50
51    return m; // Can be populated with pretrained weights
52 }
```

Listing 6. VGG16 model implementation with CRYPTDOUGH’s API

C Custom parallelization example

Listing 7 shows a baseline implementation of the oblivious control from Fig. 4b using CRYPTDOUGH’s vectorized `cmp-swp()` primitive (§5). Without virtual vectors, library developers would need to write custom code to handle operator-specific task-to-worker assignment (lines 22–39). Reusing low-level vectorized primitives, such as `cmp-swp()`, would also require “packing” data in temporary vectors (lines 8–11) and copying results back into the original vectors (lines 15–18). CRYPTDOUGH’s virtual vectors allow developers to express the same parallelization logic, without redundant copies, in three lines of code, as shown in Listing 5 of the main paper.

```

1 // Worker's task
2 void w-task(BSharedVector<T>& v, int start, int end,
3            int d, int stride, int num_ops) {
4     int n = v.size();
5     BSharedVector tmp1(num_ops);
6     BSharedVector tmp2(num_ops);
7     // Copy comparison elements into temporary vectors
8     for (int i=start, j=0; i<end && i+d<n; i+=stride, j++) {
9         tmp1[j] = v[i];
10        tmp2[j] = v[i+d];
11    }
12    // Apply secure compare-and-swap primitive (Listing 4)
13    cmp-swp(tmp1, tmp2);
14    // Copy swapped elements back to the input vector
15    for (int i=start, j=0; i<end && i+d<n; i+=stride, j++) {
16        v[i] = tmp1[j];
17        v[i+d] = tmp2[j];
18    }
19 }
20
21 // Main thread
22 BSharedVector<int> v = ... // Input vector
23 int n = v.size(), d=3, stride=2, t=NUM_THREADS;
24 int pairs = std::ceil((n-d)/stride);
25 int pairs_per_tid = std::ceil(pairs/t);
26 std::vector<std::thread> threads;
27 for (int tid=0; tid<num_threads; tid++) {
28     // Compute worker's partition
29     int start = tid * pairs_per_tid * stride;
30     int end = std::min(start + pairs_per_tid*stride, n-d);
31     if (start < n-d) { // Assign task
32         threads.emplace_back(w-task, std::ref(v),
33                               start, end, d, stride,
34                               pairs_per_tid);
35     }
36     for (auto& thread : threads) {
37         thread.join();
38     }
39 }

```

Listing 7. Example implementation of the oblivious control flow from Fig. 4 that showcases the challenges of modularity and parallelization in the absence of CRYPTDOUGH’s virtual vector abstractions.

D Core access patterns in CRYPTDOUGH

Table 6 summarizes the core data access patterns used in CRYPTDOUGH, along with their mapping functions and equivalent C++ implementations. Listing 12 shows aliases for common access patterns, and Fig. 9 provides concrete examples. The core access patterns can be composed arbitrarily to express complex oblivious control flows that the system runtime can parallelize automatically, as explained in §5.3.

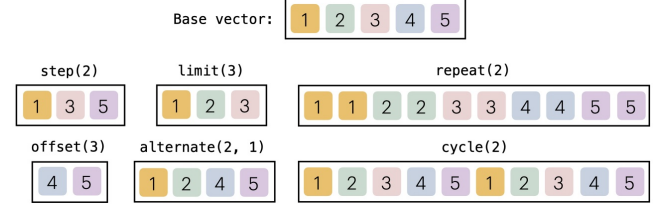


Figure 9. Examples of core data access patterns in CRYPTDOUGH

```

1 auto y = x.simple(offset, step, end);
2 int i = 0;
3 for(int f_i=offset; f_i<end; f_i+=step){
4     /* f_i = offset + step * i */
5     /* y[i] = x[f_i] */
6     i++;
7 }

```

Listing 8. C++ equivalent code for the simple pattern

```

1 auto y = x.repeated(n);
2 int i = 0;
3 for(int f_i=0; f_i<x.size(); f_i++){
4     for(int i1 = 0; i1<n; i1++){
5         /* f_i = i/n */
6         /* y[i] = x[f_i] */
7         i++;
8     }
9 }

```

Listing 9. C++ equivalent code for the repeated pattern

```

1 auto y = x.cyclic(n);
2 int i = 0;
3 for(int i1=0; i1<n; i1++){
4     for(int f_i = 0; f_i<x.size(); f_i++){
5         /* f_i = i%n */
6         /* y[i] = x[f_i] */
7         i++;
8     }
9 }

```

Listing 10. C++ equivalent code for the cyclic pattern

```

1 auto y = x.alternating(m, k);
2 int i = 0;
3 for(int i1=0; i1<x.size(); i1+=(k+m)){
4     for(int f_i=i1; f_i<i1+k && f_i<x.size(); ++f_i){
5         /* f_i = (i/m) * (m+k) + i%m */
6         /* y[i] = x[f_i] */
7         i++;
8     }
9 }

```

Listing 11. C++ equivalent code for the alternating pattern

```

1 x.offset(i) ⇔ x.shift(i,1,x.size());
2 x.step(k) ⇔ x.shift(0,k,x.size());
3 x.limit(s) ⇔ x.shift(0,1,s);
4 x.slice(i,s) ⇔ x.shift(i,1,s);
5 x.limit(s).offset(i).step(k) ⇔ x.shift(i,k,s);

```

Listing 12. Access patterns aliases used in CRYPTDOUGH

E CRYPTDOUGH protocol layer interface

Table 5 shows the interface that an applied cryptographer must implement to add a new MPC protocol to CRYPTDOUGH.

Functionality	Description
void add_a (const EVector& x, const EVector& y, EVector& z)	Secure addition ($z = x + y$)
void multiply_a (const EVector& first, const EVector& second, EVector& result)	Secure multiplication ($z = x \cdot y$)
void xor_b (const EVector& x, const EVector& y, EVector& z)	Secure bitwise XOR ($z = x \oplus y$)
void and_b (const EVector& first, const EVector& second, EVector& result)	Secure bitwise AND ($z = x \wedge y$)
void b2a (const EVector& in, EVector& out)	Boolean-to-Arithmetic conversion
void a2b (const EVector& in, EVector& out)	Arithmetic-to-Boolean conversion
std::vector<EVector> get_shares_a (const Vector& data)	Create arithmetic shares
std::vector<EVector> get_shares_b (const Vector& data)	Create boolean shares
Vector open_shares_a (const EVector& shares)	Open arithmetic shares
Vector open_shares_b (const EVector& shares)	Open boolean shares

Table 5. Adding a new MPC protocol to CRYPTDOUGH requires implementing a short list of low-level functionalities for creating, exchanging, and manipulating secret shares. All functionalities are vectorized by design, i.e., they are implemented on top of the EVector and Vector abstractions. User programs are compiled against protocol functionalities and inherit their security guarantees.

Pattern	Mapping Function	Description
Simple	$f(i) = \text{offset} + i \cdot \text{step}$ where $0 \leq i < \lceil \frac{\text{limit} - \text{offset}}{\text{step}} \rceil$	- Iterates over elements with a step, starting from an optional offset and up to an optional limit. - Useful for selecting subsets (e.g., adjacent elements). - Equivalent to a single for-loop shown in Listing 8.
Repeated	$f(i) = \lfloor i/n \rfloor$ where $0 \leq i < n \cdot V $	- Repeats each element n times consecutively, expanding the vector by a factor of n. - Useful for operations between vectors and single values. - Equivalent to an outer for-loop in nested for-loops, as shown in Listing 9.
Cyclic	$f(i) = i \bmod V $ where $0 \leq i < n \cdot V $	- Repeats the vector n times cyclically. - Used in certain join operations. - Equivalent to an inner for-loop in nested for-loops, as shown in Listing 10.
Alternating	$f(i) = \lfloor i/m \rfloor \cdot (m + k) + (i \bmod m)$ where $0 \leq i < \lfloor V /(m + k) \rfloor \cdot m + \min(m, V \bmod (m + k))$	- Alternates between including m elements and excluding k elements. - used in sorting networks (e.g., bitonic sort). - Equivalent to two fused for-loops in nested for-loops. For instance, a for-loop index is initialized using a higher level for-loop index, as shown in Listing 11.

Table 6. A virtual vector W is a restricted view of a base vector V such that $W[i] = V[f(i)]$. This table summarizes the core access patterns provided by CRYPTDOUGH to create virtual vectors, along with their mapping functions. The equivalent C++ implementations are given in Listings 8–11. Virtual vectors can be defined on top of other virtual vectors to express more complex control flows. For example, the cyclic and simple patterns can be combined to cyclically traverse every other element of a base vector.

F Additional experiments

F.1 Scalability of core primitives

We evaluate the scalability of CRYPTDOUGH’s secure primitives for the ABY, Fantastic Four (F4), and SPDZ2k protocols. Results for ABY3 are given in the main paper (Fig. 8). We use vectors with 2^{28} 64-bit elements and run multiplication, comparison ($\geq$), boolean addition, and convolution in bm-LAN. Figures 10, 11, and 12 show the execution times as we increase the number of threads per party from 1 to 32. For SPDZ2k, we never use RCA/PPA on boolean shares; therefore, we report performance for addition of arithmetic shares.

Primitive	LAN				WAN			
	ABY	ABY3	F4	SPDZ2k	ABY	ABY3	F4	SPDZ2k
Addition / XOR	627	305.7	190.4	141.0	466.1	253.0	171.7	127.0
Multiplication / AND	111.8	167.6	67.4	42.27	14.9	19.19	12.8	9.9
Comparison	12.6	29.8	9.42	0.13	4.0	7.0	3.1	0.072

Table 7. Throughput of CRYPTDOUGH’s secure primitives in millions of operations per second for the ABY, ABY3, Fantastic Four (F4), and SPDZ2k protocols.

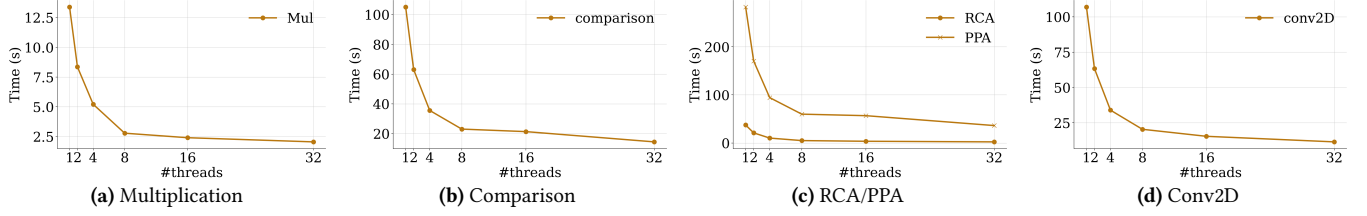


Figure 10. CRYPTDOUGH's performance as we increase the number of worker threads per party (ABY, bm-LAN).

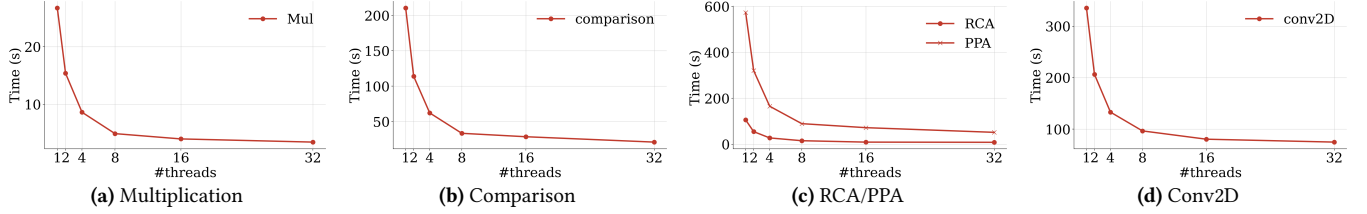


Figure 11. CRYPTDOUGH's performance as we increase the number of worker threads per party (Fantastic Four, bm-LAN).

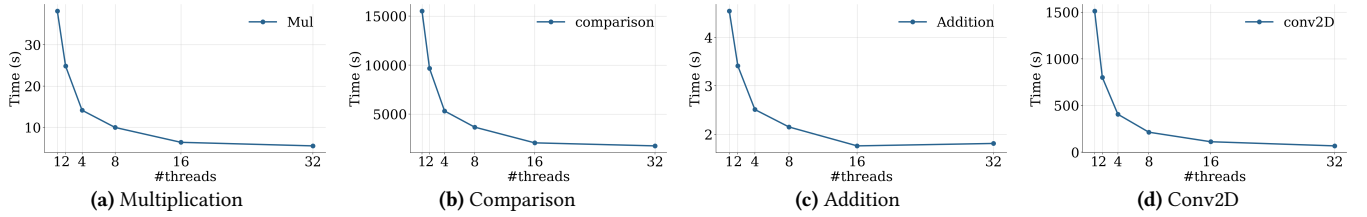


Figure 12. CRYPTDOUGH's performance as we increase the number of worker threads per party (SPDZ2k, bm-LAN).

F.2 Throughput of core primitives

Table 7 shows the throughput of CRYPTDOUGH's core secure primitives: addition/XOR, multiplication/AND, and comparison ($\geq$), for all MPC protocols supported by the system. We

measure throughput in bm-LAN and bm-WAN using 32 worker threads per party and vectors with 2^{28} 64-bit elements. Reported numbers are millions of operations per second.