HACCLE: Metaprogramming for Secure Multi-party Computation

Yuyan Bao[*][†]
yuyan.bao@uwaterloo.ca
University of Waterloo
Canada

Kirshanthan Sundararajah[*]
ksundar@purdue.edu
Purdue University
USA

Raghav Malik
malik22@purdue.edu
Purdue University
USA

Qianchuan Ye
Christopher Wagner
Nouraldin Jaber
{ye202,wagne279,njaber}@purdue.edu
Purdue University
USA

Fei Wang
Mohammad Hassan Ameri
Donghang Lu
{wang603,mameriek,lu562}@purdue.edu
Purdue University
USA

Alexander Seto
Benjamin Delaware
Roopsha Samanta
{aseto,bendy,roopsha}@purdue.edu
Purdue University
USA

Aniket Kate
Christina Garman
Jeremiah Blocki
{aniket,clg,jblocki}@purdue.edu
Purdue University
USA

Pierre-David Letourneau
Benoit Meister
Jonathan Springer
{letourneau,meister,springer}@reservoir.com
Reservoir Labs
USA

Tiark Rompf
Milind Kulkarni
{tiark,milind}@purdue.edu
Purdue University
USA

Abstract

Cryptographic techniques have the potential to enable distrusting parties to collaborate in fundamentally new ways, but their practical implementation poses numerous challenges. An important class of such cryptographic techniques is known as Secure Multi-Party Computation (MPC). Developing Secure MPC applications in realistic scenarios requires extensive knowledge spanning multiple areas of cryptography and systems. And while the steps to arrive at a solution for a particular application are often straightforward, it remains difficult to make the implementation efficient, and tedious to apply those same steps to a slightly different application from scratch. Hence, it is an important problem to design platforms for implementing Secure MPC applications with minimum effort and using techniques accessible to non-experts in cryptography.

In this paper, we present the HACCLE (High Assurance Compositional Cryptography: Languages and Environments) toolchain, specifically targeted to MPC applications. HACCLE contains an embedded domain-specific language Harpoon, for software developers without cryptographic expertise to write MPC-based programs, and uses *Lightweight Modular Staging* (LMS) for code generation.

Harpoon programs are compiled into acyclic circuits represented in HACCLE's Intermediate Representation (HIR) that serves as an abstraction over different cryptographic protocols such as secret sharing, homomorphic encryption, or garbled circuits. Implementations of different cryptographic protocols serve as different backends of our toolchain. The extensible design of HIR allows cryptographic experts to plug in new primitives and protocols to realize computation. And the use of standard metaprogramming techniques lowers the development effort significantly.

We have implemented Harpoon and HACCLE, and used them to program interesting applications (*e.g.*, secure auction) and key computation components of Secure MPC applications (*e.g.*, matrix-vector multiplication and merge sort). We show that the performance is improved by using our optimization strategies and heuristics.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**.

Keywords: metaprogramming, domain-specific language, secure multi-party computation

ACM Reference Format:
Yuyan Bao, Kirshanthan Sundararajah, Raghav Malik, Qianchuan Ye, Christopher Wagner, Nouraldin Jaber, Fei Wang, Mohammad Hassan Ameri, Donghang Lu, Alexander Seto, Benjamin Delaware, Roopsha Samanta, Aniket Kate, Christina Garman, Jeremiah Blocki,

[*] Both authors contributed equally to this work.
[†] Work performed while author was at Purdue University.

GPCE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9112-2/21/10.
https://doi.org/10.1145/3486609.3487205

Pierre-David Letourneau, Benoit Meister, Jonathan Springer, Tiark Rompf, and Milind Kulkarni. 2021. HACCLE: Metaprogramming for Secure Multi-party Computation. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486609.3487205

1 Introduction

Secure Multi-Party Computation (MPC) enables a group of distrusting parties to jointly perform computation without revealing any participant's private data that they do not wish to share with others. It has broad practical applications, *e.g.*, Yao's millionaires problem [50], secure auctions [5, 24], voting, privacy-preserving network security monitoring [8], privacy-preserving genomics [26, 49], private stable matching [18], ad conversion [28], spam filtering on encrypted email [22], and privacy-preserving machine learning [16]. Secure MPC applications are generally realized as circuits communicating information – both private and public – among parties.

Although MPC techniques and protocols have seen much success in the cryptography community, it is still challenging to build practical MPC applications. Executing cryptographic protocols is notoriously slow, due to the encryption and communication overhead. The largest benchmark reported in Fairplay [33] – a secure two-party computation system – was finding the median of two sorted input arrays containing ten 16-bit numbers from each party. Running the benchmark required execution of 4383 gates, and took over 7 seconds on a local area network. While improving computing capabilities and network bandwidth, implementation techniques can contribute to 3-4 orders of magnitude improvements [20]. These techniques include optimizations that reduce the number of gates and the depth of a circuit and reduce the computational costs of executing a cryptographic protocol. However, such optimizations do not exist in general-purpose compiler frameworks.

While several MPC frameworks have been proposed [4, 9, 15, 17, 25, 29, 31, 32, 35, 43, 47, 48, 51, 51, 52], they either provide low-level cryptographic primitives or high-level abstractions like traditional programming languages, but not both. The low-level frameworks provide high degrees of customized protocol execution, but the users are generally expected to be experts in either one or both of cryptography and optimizing circuits. These MPC frameworks provide little or no type safety to prevent semantic errors, and it is difficult to write applications in a way that is portable across different protocols. The high-level frameworks provide traditional programming abstractions that hide the data-oblivious nature of secure computation from programmers. But these frameworks are tied to only one or a few protocols and their compilation procedures – from high-level abstractions to low-level primitives – are not easy to extend to perform application-specific optimizations [51].

Contributions The main intellectual contribution of this paper is a toolchain for developing Secure MPC applications called HACCLE (High Assurance Compositional Cryptography: Languages and Environments). Our framework contains an embedded domain-specific language (eDSL) *Harpoon* for designing MPC-based applications, and uses standard metaprogramming techniques to lower the development effort. Allowing seamless construction of MPC-based applications by software developers without expertise in advanced cryptography is the main purpose of providing such a high-level programming language. A Harpoon program is compiled to an acyclic combinational circuit, which is described in a HACCLE Intermediate Representation (HIR). HIR exposes the essential data-oblivious nature of MPC, and allows cryptography experts to experiment with new primitives and protocols. Our framework also provides a specialized backend for estimating the resource usage (*e.g.*, compute time and memory space) prior to execution.

This paper makes the following specific contributions:

- **HACCLE Toolchain**: A compilation framework to build and execute MPC applications written in Harpoon – an embedded domain-specific language (eDSL) in Scala based on the LMS metaprogramming and compiler platform.
- **HACCLE Intermediate Representation (HIR)**: An extensible circuit-like intermediate representation tailored to abstract cryptographic primitives used in MPC.
- **Optimization Strategies**: Methods for optimizing the MPC application by specialization as it flows through each stage of our HACCLE toolchain.

The rest of the paper is organized as follows. Sec. 2 provides background on cryptographic protocols involved in Secure MPC and motivates the need for developing MPC-based applications. We describe the key impediments for developing practical MPC applications with the example of secure auctions. Sec. 3 illustrates components of our compiler and HIR. Sec. 4 describes the HACCLE toolchain and associated workflow. Sec. 5 describes the optimizations implemented in our compiler toolchain. Sec. 6 discusses our toolchain on three case studies in detail. Sec. 7 summarizes related work and Sec. 8 concludes the paper. The HACCLE implementation is available online at:
https://github.com/YuyanBao/HACCLE.

2 Motivating Example and Background

As an example of Secure MPC, consider online auctions. Online auctions have great practical importance and different models are widely used, *e.g.*, by eBay, Google AdWords, and Facebook. In general, a secure online auction works as follows. Buyers place their sealed bids on items, and for each item, the highest bidder is chosen to buy it. In this setup, parties are not permitted to know others' bids. Hence, conducting successful secret auctions in the absence of a trusted authority requires cryptographic techniques to preserve the secrecy of bids while performing necessary computation, such as finding the highest bidder, in an assuredly trustworthy way. One of the significant use cases of secure auctions is procurement via a competitive bidding process, where no participant trusts each other, including the auctioneer. While a trusted third party handling the auction may be acceptable when the items under auction have low value, this is generally a less desirable option in high-value and corruption-prone environments, such as procurement for public construction contracts.

There are many different types of auction policies studied by economists and game theorists. An auction where the highest bidder is chosen to buy the item by paying the highest bid is known as a *first-price* auction. A *second-price* or Vickrey auction [46] is an alternative auction policy where the highest bidder is chosen to buy the item at the *second* highest price. Second-price auctions provide buyers with the incentive to bid their true valuation and do not allow for price discovery (*i.e.*, no ramping up of prices). Hence, second-price auctions are especially suitable for high-value low-trust environments, such as public procurement. Second-price auctions also apply to settings where multiple items are auctioned and/or bids may have additional structure, such as if/then conditions to evaluate specific contract terms that need to be taken into account for comparison. Such settings are described as generalized second-price auctions. Given that secrecy of the bids is preserved, the computation required when a single item is auctioned is simpler than when multiple items are auctioned. Hence it is desirable both from programmability and efficiency viewpoints that the online auction application is written once for the general case and gets automatically and correctly specialized for the desired number of items, number of bidders, comparison logic, etc.

Most implementation techniques for Secure MPC applications (*e.g.*, first- and second-price auctions) are based on circuits. Equivalent functionality can be expressed as a Scala program: *e.g.*, the following expresses an AND gate template, with bit-width determined by the input array:

```
val input = Array(0, 1, 1, 0)
var res = input(0)
for (i <- (1 until input.length))
  res = res & input(i)
res
```

Just like in DSLs for hardware design [2, 27], using metaprogramming techniques to *stage* bitwise operations rather than execute them directly is the key to our approach. Implementing *secure* circuits then amounts to specializing the encoding and operators for the respective cryptographic backends.

We use Lightweight Modular Staging (LMS) [38] to turn the encoding and operators into *staged* expressions, so that programs like the previous AND template become circuit generators. In LMS, type constructor Rep[T] is used to denote a *staged* expression, which will cause an expression of type T to become part of the generated program. The following code shows the high-level design of HACCLE intermediate representation (HIR) using LMS. The case classes Bit and Num define the primitive constructs of encoding boolean circuits and arithmetic circuits respectively, where the types Rep[SBit] and Rep[SNum] denote the staged representations of secure bits and numbers (see Sec. 3.3).

```
// Boolean circuit interface
abstract class SBit
case class Bit(val value: Rep[SBit], ... ) {
  def &(that: Bit) = { ... }
  def |(that: Bit) = { ... }
}
// Arithmetic circuit interface
abstract class SNum
case class Num(val value: Rep[SNum], ...) {
  def +(that: Num) = { ... }
  def -(that: Num) = { ... }
  def <(that: Num) = { ... }
}
```

It is of course possible to implement Num on top of binary circuits and Bit arrays using standard half adders and full adders (see Sec. 3.3), but some secure cryptographic protocols directly support arithmetic circuits.

Now to express a secure first-price auction, we can use operations on an array of pairs of Nums that denote encrypted bidders' identities and their bids:

```
// assume input: Array[(Num, Num)]
var res = input(0)
for (i <- (1 until input.length))
  res = if (res._2 < input(i)._2) input(i) else res
res
```

Observe that the linear sequence of operations in the above code results in a suboptimal circuit. Rewriting the code in a functional style, as, input.reduce(_ max _), allows us to abstract over the reduction pattern and substitute the linear sequence with a tree reduction patten, which yields a circuit of logarithmic depth, allowing efficient parallel computation. Using known techniques for extracting functional dependencies from imperative loops [19, 37], this transformation is automated and applied to for loops. Now, all we need are generic functions: max, sndmax (shown below) and reduce (Fig. 1). The latter divides the computation into subproblems of size $n/2$ and call the subproblems recursively.

```
// compare (bid id, bid value)
def max(a: (Num, Num), b: (Num, Num)): (Num, Num) =
  if (a._2 < b._2) b else a
// compare (bid id, bid value, price = 2nd highest bid)
def sndmax(a: (Num, Num, Num), b: (Num, Num, Num)) =
  val prz = ... // 2nd highest of a._2,a._3,b._2,b._3
  if (a._2 < b._2) (b._1,b._2,prz) else (a._1,a._2,prz)
```

Type classes, *e.g.*, Ordering[T] or Encoding[T], can be used to further abstract over comparison or access logic.

With the given comparator functions, we can transform the previous imperative code to a functional style, which generates optimal circuits:

```
def reduce[T](input: Array[T])(f: (T, T) => T): T = {
  def rec(elems: Array[T]): T =
    if (elems.length == 1) elems(0)
    else {
      val b1 = elems.slice(0, elems.length/2)
      val b2 = elems.slice(elems.length/2, elems.length)
      f(rec(b1), rec(b2))
    }
   rec(input)
}
```

Figure 1. Generic function reduce yielding a circuit of logarithmic depth.

```
// compute first-price auction
val max = reduce(input)(max)
// compute second-price auction
def initPrice(x) = (x._1, x._2, x._2)
def dropSecretBidValue(x) = (x._1, x._3)
val r = reduce(input.map(initPrice))(sndmax)
val snd_max = dropSecretBidValue(r)
```

For second-price auctions, we transform each element in the array to a 3-tuple of bidder's identity, highest bid, and initial price, and reduce with sndmax. Sec. 6.1 shows a full implementation in our HACCLE toolchain.

We continue our discussion of Secure MPC background by looking at different protocols for secure computation, system models, trust models, and the offline/online paradigm.

Secret sharing. Secret sharing [40] is a cryptographic technique that distributes secret data amongst a group of parties, and allows the secret to be reconstructed only when a sufficient portion of shares are combined. A (t, n)-secret sharing scheme allows the secret s to be split into n shares. Any $t - 1$ of the shares reveal no information about s, while any t shares can complete reconstruction of the secret s.

The SPDZ [15] and HoneyBadgerMPC [32] frameworks serve as our secret sharing backends and provide Python-style programming environments for writing custom MPC programs. These frameworks let developers express MPC programs (*e.g.*, second-price auction) as arithmetic expressions. Constructing the most efficient MPC programs is the major challenge for developers. First, developers must know how to build an efficient circuit, *e.g.*, realizing a balanced tree reduction to reduce the depth of a circuit and to parallelize the computation, instead of performing a linear reduction over a list of elements. Second, developers must have a good understanding of the cost of every primitive operation (*e.g.*, usage of logically similar but different comparison operators may yield different costs). These challenges are significantly different from writing an efficient program in the traditional setting and can be successfully overcome by a compiler.

Homomorphic Encryption. Cloud computing may violate privacy. In this scenario, one party wants to perform computation by outsourcing to another (possibly untrusted) party, *e.g.*, training machine learning models of private data on a public cloud server. This can be achieved by homomorphic encryption, another important cryptographic primitive. *Homomorphic encryption* enables operations on encrypted data. The PALISADE [44], TFHE [13], and HElib [39] libraries serve as our FHE backends. They all implement asymmetric protocols that use a pair of public and private keys for encryption and decryption. The TFHE library implements a very fast gate-by-gate bootstrapping mechanism [11, 12], and allows to evaluate a boolean circuit composed of binary gates over encrypted data. The HElib library implements many optimizations to make homomorphic evaluation run faster. The PALISADE library supports the BGV [7], BFV [6, 21], and CKKS [10] schemes. In cryptography, ciphertext and plaintext mean private and public information, respectively. In this paper, we may use these terms interchangeably.

Garbled Circuits. Yao's Garbled circuits [50] is a two-party secure computation scheme for boolean circuits against semi-honest adversaries. Obliv-C [51] is the library that we use to support Yao's Garbled Circuits protocols.

System and Communication Models. There are two popular system models for multi-party computation. The MPC-as-a-service setting allows some parties to play the role of servers and to provide MPC services to clients with private input. The other setting is where the parties running the MPC protocols are the participants who provide the input. The HACCLE toolchain does not enforce a specific setting; instead, users choose the suitable setting for their applications and keep that setting in mind when developing programs. Similarly, the HACCLE toolchain does not enforce any communication model. The parties/machines could be fully connected, could form a star network structure, or could be any specified structure. As long as the network structure is supported by one of HACCLE's backends, HACCLE is able to compile the program.

Trust/Adversary Models. Developing MPC applications requires understanding the security assumptions of an MPC library, such as the trust/adversary models. There are two major adversary models: semi-honest and malicious. A *semi-honest* adversary follows the protocol, but tries to learn from received messages. A *malicious* adversary has the same power as a semi-honest one in analyzing the protocol execution. In addition, it may also control, manipulate, or arbitrarily inject messages to the network. In HACCLE, programmers only need to provide a model of choice and the toolchain will pick proper sub-protocols to build up the MPC programs satisfying the adversary model described.

Offline Phase. The offline/online paradigm is applied by many MPC protocols and frameworks. The online phase uses a buffer of preprocessed input-independent values created during the offline phase. Thus, the MPC framework can run the offline phase to prepare them beforehand. The online phase is where clients/users provide their inputs and get expected output; it can gain a significant speed-up with the help of the offline phase. A number of preprocessed values are

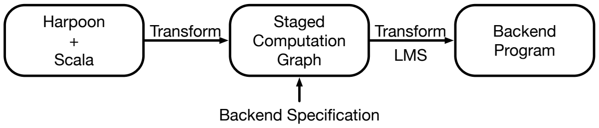

Figure 2. Compilation in HACCLE.

required for multiplications and comparisons. The volume of preprocessing data depends on the online phase, and it is hard for programmers without security expertise to work out those requirements. In HACCLE, programmers need not care about the secret parameters. They describe only the computation and the private information. The HACCLE toolchain can synthesize suitable settings for the offline phase.

3 Compiler

The HACCLE toolchain uses LMS [38] to support our towers of abstractions. Staging is a technique for building extensible, flexible DSLs by providing *code generators* that successively lower higher-level abstractions to lower-level abstractions, and, ultimately, to executable code. Importantly, staging allows optimization to be performed at every level of the lowering process. Hence, some optimizations can be performed at high levels of abstraction (*e.g.*, optimization on plaintext computation (see Sec. 5)), while other optimizations can be performed at lower levels of abstraction. As a result, abstraction penalties are minimized. Another benefit of staging is that because the translation is written in terms of generators, it is simple to add new abstractions at any given level.

3.1 Staged Compilation

Multi-Stage Programming [42] (or staging) is the programming language technique that executes programs in multiple stages. A staged computation does not immediately compute a result, but returns a program fragment that represents the computation and that can be explicitly executed to form the next computational stage. The key benefit of staging is that the present-stage code can be written in a high-level style, yet generates future-stage code that is very low-level and efficient. Fig. 2 illustrates an end-to-end compilation path in HACCLE. The compiler takes a Scala program with Harpoon annotations (see Sec. 3.2), and constructs a computation graph that expresses an abstract circuit. Given a backend specification, the compiler will generate a target program for it. Currently, our compiler is not able to automatically choose an appropriate backend and initialize all the parameters for it. Thus, a backend specification is needed. It is a file that contains a set of parameters for translating an abstract circuit to a concrete backend program.

Generative Programming and Lightweight Modular Staging (LMS). As mentioned in Sec. 2, the HACCLE compiler uses LMS for code generation due to its metaprogramming capabilities, and the type constructor `Rep[T]` is used to denote a staged expression. For example, the type `Rep[SNum]` denotes an encrypted integer. Given two `Rep[SNum]` values `a` and `b`, *evaluating* the expression $a + b$ will *generate* code for a given backend. For the Helib backend, the generated code will be `Ctxt r = a; r += b;`, where `Ctxt` is the type of a ciphertext in the Helib library. For the TFHE backend, the generated code will be:

```
LweSample* x5 =
new_gate_bootstrapping_ciphertext_array(64, x2->params);
fhe_add(x5, a, b, 64, bk);
```

where `LweSample` is the type of a ciphertext in the TFHE library. As a TFHE program does not provide arithmetic expressions and operations, the compiler encodes an integer as a bit-array of size 64. The function `fhe_add` is part of our HACCLE library of the TFHE library.

3.2 Harpoon

HAccle Rich Representation for Program OperatiON (*Harpoon*) language is an expressive subset of Scala for writing MPC programs. It is an imperative and monomorphic language, featuring standard control flow operations: loops, function calls, conditionals, and recursions. The language is designed to be expressive enough that programmers could easily write Harpoon code directly, while being constrained enough to ensure that Harpoon programs can be implemented via translation to secure low-level computation. In practice, Harpoon serves as the top-level IR for the HACCLE pipeline, and is the language for end-user programs.

The Harpoon language is not only able to access Scala libraries, but also provides a set of cryptographic data structures, *e.g.*, `HArray[T]` is an encrypted array that allows one to index on ciphertexts. It also provides a set of security annotations that are read via reflection and are used to direct code generation. They are agnostic to the target backend, and are used by subsequent stages of the HACCLE pipeline. For example, the annotation `sec` is used to mark the provider (also the owner) of private data. Recursive functions and loops may be annotated with an upper bound on the number of recursive calls and iterations. This expression can reference the parameters of the function, allowing this bound to vary according to the context where a function is called, *e.g.*, consider the signature of `merge` function:

```
@bound(a.length + b.length)
def merge(a: HArray[Int], b: HArray[Int]): Harray[Int]
```

The upper bound of the number of recursive calls is the sum of the length of the two input arrays. Note that the semantics of function calls in Harpoon is not impacted by the bound; rather it is used by subsequent stages of the pipeline to bound the invocation of a recursive function call (see Sec. 3.4).

The annotated program is also equipped with a type system, and ensures that information about private data cannot be leaked. This provides the first-layer guarantees that the programs can be successfully compiled by the later stages of the pipeline. Consider the statement `println(a)`, where *a* is annotated as private data. The compiler will report a type error, as encrypted data is not understandable or meaningful

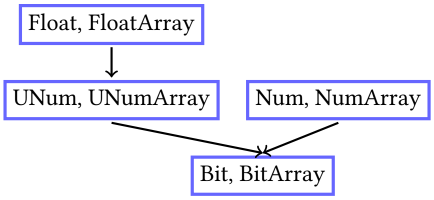

Figure 3. Example of multi-level HIRs.

to users. But the assignment `@sec(alice) val r = a` is permitted, as the annotation expresses that the variable `r` stores encrypted data. While the type system at this stage does not make use of fine-grained ownership information, this information will be passed down through the pipeline. See [3] for the details of Harpoon language.

3.3 Intermediate Representation

HACCLE intermediate representation (HIR) serves as an interface between high-level programming languages and cryptographic backends. HIR is a domain-specific intermediate language, and gains benefits from LMS to support towers of abstractions. It encompasses all the primitive operations which we have supported so far, *e.g.*, encryption, decryption, sharing, and combining.

Multi-level IR. Different backends may support different sets of operations in HIR—no backend is "complete" in that there is a direct implementation of each HIR operation in that backend. For example, the TFHE backend supports logical operations but not arithmetic ones. In contrast, other backends may support arithmetic operations but not boolean ones. The compiler's job is to *rewrite* HIR circuits to be compatible with backends.

As shown in Fig. 3, HIR is a multi-level IR. The compiler can thus use rewrites to target the subset of operations that a given backend supports. For example, arithmetic operations (adds, multiplies) can be rewritten into bit-level implementations (as, *e.g.*, ripple-carry adders, or bit-level implementations), or boolean operations can be represented as arithmetic operations that happen to operate over $\mathbb{Z}_2$. We have developed a set of these rewrite rules for various backends (and, indeed, rely on exactly this type of rewrite to support floating point operations).

A key task for integrating a new backend is identifying what set of HIR operations that module supports, hence directing the compiler to perform appropriate rewrites. Notably, if the compiler *cannot* rewrite an HIR circuit to target the set of operations a backend supports, it will manifest as a type error, providing feedback to the user.

In the scenario of using a FHE scheme, an integer is encoded as the `Num` data structure shown below, where the fields `provider` and `value` are abstraction of the party who provides the value and the encrypted value respectively.

```
case class Num(
  val provider: Set[Rep[SOwner]], // who provides it
  val value   : Rep[SNum]         // encrypted value
)
```

In this case, a variable declaration statement in Harpoon, *i.e.*, `@sec(alic) val x = 5;`, is transformed to `val o = new Owner(alice); val x = Num(o, 5);` in HIR.

In the scenario of using a secret sharing scheme, an integer is encoded as the `ShareNum` data structure in HIR shown below. It expresses a general secret sharing protocol. The `provider` is the one who contributes the `value` that is shared among a set of `players` with `threshold`. The set of `observers` are those who are allowed to access the value once it gets combined.

```
case class ShareNum(
  val provider  : Set[Rep[SOwner]], // who provides it
  val players   : Set[Rep[SOwner]], // players
  val observers : Set[Rep[SOwner]], // who observes it
  val threshold : Int,              // threshold
  val value     : Rep[SShareNum]    // shares
)
```

In addition, HIR provides libraries for implementing secure computation. Those libraries are not supported by general-purpose compilers, but are essential to build interesting multi-party applications with security guarantees. For example, the following shows the operations of an array supporting indexing on a ciphertext, where `arr` is an HIR array.

- `arr(i)`: array index, where *i* is a plaintext or a ciphertext.
- `arr.update(i, v)`: update the *i*th element with the value *v*, where *i* is either a plaintext or a ciphertext.
- `arr.slice(i, j)`: array slicing from the *i*th element until the *j*the element, where *i* and *j* are plaintext.
- `arr.length`: the length of the array

The way these array operations with secure indices are currently implemented is through, essentially, a naive Oblivious RAM (ORAM): to index into an array with a ciphertext index, the compiler generates a circuit that wires every array element, and a secure selector (multiplexer) to output the desired array element. This is equivalent to a set of *if-then-else*s to choose the desired array element, except with a logarithmic depth instead of a linear depth. Writing to an array element with a ciphertext index is the equivalent of an array copy, where each element of the new array performs a check for whether the old element of the array should be copied, or the "update" value should be copied.

As implementation details of cryptographic backends are abstracted away from HIR, our framework can be easily extended to support more advanced cryptographic backends, for example, a backend with ORAM. Here, we would leverage HIR's ability to provide backend-specific rewrite rules, and would directly rewrite array operations to ORAM operations.

Type System. HIR also abstracts away the implementation details of cryptographic primitives and protocols. For example, an addition operation does not specify how a secure addition is achieved, as different protocols perform in different ways. But the type rules provide an approximation of data access policy that specifies how data is provided, accessed, and shared. For example, an addition operation on two shared numbers is only allowed on the same set of players with the same threshold, which are known at compile time. And the result is provided by either one of its operand's providers with the same set of players with the same threshold, and is allowed to be accessed by either one of the operands' observers.

```
def +(x: ShareNum, y: ShareNum) = {
 assert(x.players.equals(y.players))
 assert(x.threshold == y.threshold)
 ShareNum(x.provider | y.provider, players,
 x.observers | y.observers, threshold, value.+(y.value))
}
```

Given a cryptographic backend, HIR code is further transformed to a program with the corresponding cryptographic semantics. And the HIR type system is refined to provide more precise information on data access policy. For example, the type rule of the addition operation is refined to the following when using the additive secret sharing scheme.

```
def +(x: ShareNum, y: ShareNum) = {
 assert(x.players.equals(y.players))
 assert(x.players.size == x.threshold)
 assert(x.threshold == y.threshold)
 ShareNum(x.provider | y.provider, players,
 x.observers & y.observers, threshold, value.+(y.value))
}
```

The type rule checks it is a *n*-out-of-*n* secret sharing scheme, *i.e.*, `x.players.size == x.threshold`. The refined type rule provides a stronger security guarantee, *i.e.*, the transformed program is compatible with the semantics of the backend. For example, an FHE target program is not transformed to a program that may invoke secret sharing primitives. See [3] for the details of HIR.

3.4 Obliviousness

In addition to bridging the semantic gap between a high and a low-level language, our compiler also bridges the semantic gap of obliviousness. A program without privacy concern diverts its control flow according to the input: statements are executed conditionally, loop for a variable number of iterations, etc. To protect privacy, boolean and arithmetic circuits have to be oblivious in the sense that they perform the same sequence of operations regardless of the input. The following transformations may seem quite inefficient at first sight, but they are absolutely necessary in order to maintain obliviousness.

Encrypted Array Indexing. Indexing an array with a ciphertext is encoded as a multiplexer circuit that takes every element of the array as an input and outputs the element in the position. This multiplexer circuit consists of integer comparators and selectors.

Conditional Execution. After a typed Harpoon program is transformed to HIR code, there are two types of if-constructs allowed. One is the standard if-statement, where its condition depends on plaintext comparisons, and the two branches consist of a sequence of statements that may have side effects. The other has the form `z = if (b) x else y`, where the value of `b` is the result of private comparisons. Obliviousness is effectively guaranteed by executing both the consequent and alternative branches. If the backend is a boolean circuit, this if-construct is further transformed to a selector. If the backend is an arithmetic circuit, the program is transformed to `z = b * x + (1 - b) * y`. In the following Harpoon code snippet, the variable `arr` stores a sequence of shared numbers, and the comparison result of `max < arr(i)` is a shared secret value. Thus, the program

```
if (max < arr(i)) { max = arr(i) }
```

is transformed to

```
val b = max < arr(i)
max = b * arr(i) + (1 - b) * max
```

Note that such a program transformation is non-trivial for a program allowing mutable states. Currently, an if-statement will be transformed if the side effects of its two branches can be syntactically detected.

Loops and Recursion. All function calls are treated as macros and are simply inlined. All loops are unfolded as the number of iterations is a compile-time constant. Fig. 4 demonstrates our treatment of recursive calls, where the obliviousness is achieved by using the extra plaintext parameter `d` on the right side of the figure. In the transformed program, the value `d` is initialized by the Harpoon annotation and decreases with each iteration. This makes sure that the recursive call only iterates `d` times. Note that the function `func` is a polymorphic overloading function in HIR.

3.5 Code Generation

Cryptographic Backends. In the context of building circuits, LMS is used to specialized a circuit with respect to a target backend. The outcome of such a programmatic specialization is a compiled target of the circuit. The code generator transforms an abstract circuit to a concrete one for a given backend. For example, the following adder expressed in HIR is specialized to a boolean or arithmetic circuit based on the backend.

```
val o1 = Owner();
output((Num(o1, 10).+(Num(o1, 5))).eval(o1))
```

The essence of multi-stage programming is to generate efficient programs using high-level constructs without run-time penalty [41]. The example in Fig. 5 a shows a code snippet that generates a for loop. Note that the `if` condition is composed of a plaintext boolean type, so this code is executed at code generation time as shown Fig. 5 b.

Resource Estimation. This is one of the special noteworthy backends: instead of performing a computation, it generates a graphical representation of the HIR circuit, which is

Scala Program

```
val a = 5
val b = 15
def gcd(x: Int, y: Int)
  : Int = {
  if (x == 0) y
  else gcd(y % x, x)
}
println(gcd(a, b))
```

Harpoon Program

```
@sec(alice) val a = 5
@sec(alice) val b = 15
@bound(5)
def gcd(x: Int @sec, y: Int @sec)
  : Int @sec = {
  if (x == 0) y
  else gcd(y % x, x)
}
@reveal(alice) val r = gcd(a, b)
println(r)
```

HIR Program

```
val o = Owner(alice)
val a = Num(o, 5)    val b = Num(o, 15)
val gcd = func((d: Rep[Int], x: Rep[SNum],
                y: Rep[SNum]) => {
  if (d == 0) y
  else if (x == 0) gcd(d-1, x, y)
       else gcd(d - 1, y % x, x)
})
val r = Num(o, gcd(5, a.value, b.value)).eval(o)
println(r)
```

Figure 4. Compute the Greatest Common Divisor (GCD) of two numbers. The left one shows the Scala textbook implementation. The middle one shows the Harpoon program. The annotations express that a user, alice, computes the GCD of her private data a and b through a different party, which performs computation on the data in an encrypted form, and provides the encrypted results to alice. The right one shows the corresponding HIR program. The translated gcd function has one extra parameter d initialized by the bound Harpoon annotation, and decreases with each iteration.

(a) HIR code example:

```
val sum = func((x: Rep[SNumArray], len: Rep[Int]) => {
  var n = 0     val b = true
  var res = Num(o1, 0).value
  while (n < len) {
    if (b) {  res = res + x(n) }
    n += 1
  }
  res
})
```

(b) Generated C code of TFHE backend:

```
const LweSample* x3(const LweSample* x4, int x5){
 int x6 = 0;
 const LweSample* x7 = num_init(0, 64 ,x2);
 while (x6 < x5) {
  x7 = add(x7, array_index(x4, x6, 64, x0), 64, x0);
  x6 = x6 + 1;
 }
 return x7;
}
```

Figure 5. (a) HIR code example (b) Generated C code of TFHE backend.

fed to a generic "Evaluator". This is a *resource estimation program* that traverses the graph and performs analysis at each node. The estimator is parameterized on a given resource model, which specifies costs of each node, edge type, and the depth of each edge in the graph.

At the most basic level, the resource estimation framework expects an enumeration of the *abstract gates* for a particular cost model, a description of how each HIR node type affects these gates, and depths. The total cost is tallied in terms of abstract gates. For example, a cost model for a secret sharing backend may have *round complexity* and *communication complexity* as its abstract gates, whereas a circuit backend may have AND, OR, and NOT as its abstract gates. The evaluator traverses the HIR graph and accumulates the abstract gate costs produced by each node, and tracks the maximum total depth encountered for critical path estimation. In the case of a secret sharing scheme, traversing the graph will potentially increment round and communication complexity as new computation nodes are encountered, whereas a circuit backend will increment gate costs. These gate costs are then instantiated with specific costs (in terms of lower-level operations) based on the resource estimates determined by cryptographic experts.

This framework can also be easily extended to evaluate costs that do not follow this simple model. A data structure at each HIR node and a function that performs accumulation of cost based on the type of the node are sufficient to estimate the cost. Cost models can also be parameterized on values which are configurable but known at compile time (*e.g.*, integer bitwidth). The prime modulus can be determined by the security specifications, and specific edge costs. The ability to estimate the cost of a program becomes useful when selecting a target from multiple backends. A program may be better suited for execution on a particular backend than another. If the available backends' cost models are comparable, then we can generate resource estimations to choose the best one for execution.

4 HACCLE Workflow

This section describes the compilation flow of our HACCLE framework as shown in Fig. 6. In the very first stage of the flow, an input program is staged to a complete Harpoon program that consists of an entry point for the inputs provided by the parties, computation and necessary revealing

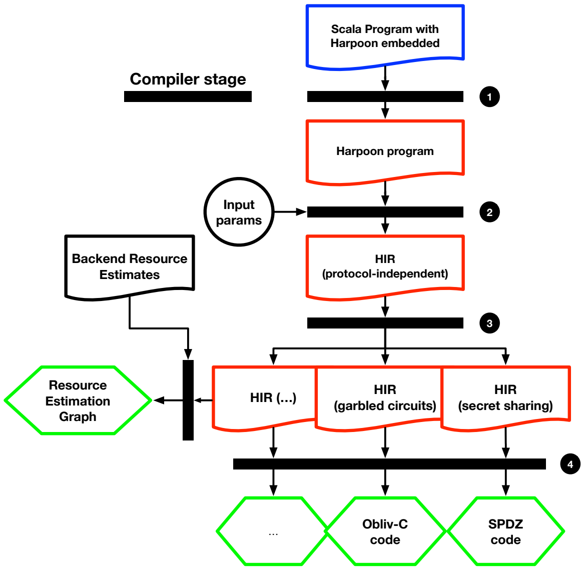

Figure 6. HACCLE Compilation Framework.

of results. The Harpoon program is compiled to HIR code, which is one big acyclic circuit, and is further lowered to the protocol specific HIR program. Finally, the code for a specific backend is generated from the low-level HIR program. Resource estimation models the resource usage of the computation in a specific protocol, and guides the compiler to generate optimal code. The following subsections illustrate the stages of our toolchain from writing an MPC application as a program to executing it using different protocols, and how the type system provides various security guarantees at different stages.

4.1 Specifying the Program

A programmer starts by providing a Scala program that embeds a secure computation, which is written in Harpoon (see Sec. 3.2). The Scala program runs at client locations, and is responsible for processing input, setting up communication channels, etc. The Harpoon program actually performs the secure computation that is written parametrically: effectively, a Harpoon program is a function that accepts the number of parties and their inputs as parameters.

4.2 Generating a Circuit

Stage 1 The first stage of compilation transforms a Scala + Harpoon program to a pure Harpoon program, *i.e.*, executing a Scala program stages away the *non-Harpoon* fragment of the code: local input files are read into memory and connections are set up to the relevant servers.

After the stage 1 compilation, a Harpoon program represents *just the secure computation that must be performed.* This program will eventually be transformed to a circuit that performs the desired secure processing. However, the secure computation is not ready for execution yet. Any publicly known information about the inputs (*e.g.*, the bitwidths, or the maximum input size) has not yet been incorporated into the circuit, and the input values are not yet known. At this stage, the Harpoon type system provides the key security guarantee that private data will not leak via public channels.

An important note is that each Harpoon program represents a single secure computation that compiles to a single circuit. Hence, the Harpoon program must compile down to a circuit whose size is determined only by the publicly available information about the inputs. In many applications, there are multiple secure computation that must occur (*e.g.*, in database applications, there may be multiple queries; each query represents a different secure computation). Here, we leverage the blurred distinction between compile time and runtime. Generating a Harpoon program happens at what programmers traditionally consider *run time*: the Scala program is *actually running* to produce the Harpoon program. Hence, the *Scala program* can include a loop over the set of queries, and for each query, a new Harpoon program is generated, compiled and executed. The abstraction in Scala has no runtime overhead for the generated code since it is executed at the Scala runtime, offering the so-called "abstraction without regret" (see Sec. 3.1).

Stage 2 The next step is to generate an *abstract circuit*: a Harpoon program is compiled down to HIR code (see Sec. 3.3), which is, essentially, a bounded-size and single-assignment representation of the program. Here, the bound annotation in the Harpoon program is used to unroll loops and inline recursive functions, leading to a functional and loop-free representation of the program. The HIR program at this stage is still independent of a particular protocol. Hence, it is essentially a direct translation of the Harpoon program into HIR code without considering the abilities of any particular backend. The key typing guarantee that HIR code provides at this level is that the appropriate HIR operation will be used based on whether inputs to an operation are private or public.

Stage 3 The next compilation stage specializes an HIR circuit to a specific protocol. The choice of protocol is determined by the security specification file. Here, we do not change the *language* representation of the program—the resulting program is still in HIR. Instead, this stage rewrites HIR code to limit the use of HIR operations to those supported by a particular backend. For example, a backend that only supports boolean operations requires translating all operations on integers and floating point to bit-level operations. Similarly, a backend that only supports operations on integers requires translating floating point operations to decomposed operations on the component parts (mantissa and exponent). Here, HIR switches to the use of backend-specific type systems that enforce the following property: a type-checked backend-specific HIR circuit enforces the requirements of that backend for security (*e.g.*, the set of sharers matches up when performing operations in a secret-sharing backend).

Stage 4 The final step of generating a circuit is specific to a backend implementation. Here, an HIR circuit is translated to be compatible with a particular backend. This is the key module interface provided by our system. It may require translating the circuit to a set of API calls (*e.g.*, our TFHE backend), or to a different programming language (*e.g.*, translating to Obliv-C for the garbled-circuit backend, or Scale-Mamba for the secret-sharing backend). The backend is configured based on the information in the security specification file. At this point, the circuit is in an executable form, and can perform the desired secure computation, using the actual inputs from the various parties.

5 Optimization

Our compiler contains a set of optimizing transformations, *e.g.*, peephole optimizations, common subexpression elimination, constant folding, and dead code elimination. In addition

Harpoon Program:

```
@sec(alice) val a = 2
scala.math.pow(2, 8)
```

HIR Program:

```
val o1 = Owner()
UNum(o1, 2).pow(8)
```

Generated TFHE program:

```
const LweSample* x3 = unum_init(2, 64, x2);
LweSample* x4 = unum_mul(x3, x3, 64, x0);
LweSample* x5 = unum_mul(x4, x4, 64, x0);
return unum_mul(x5, x5, 64, x0);
```

Figure 7. Computing $pow(2, 8)$, where 2 is private, and `x0` and `x2` are the cloud key and private key used for encryption.

to those optimizations that a general purpose compiler has, we identified several optimizations specific to Secure MPC circuits. Given an in-memory representation of a boolean or an arithmetic circuit, these optimizations reduce the depth of circuits and the number of costly gates.

5.1 Scalar Multiplication

The multiplicative depth of circuits is the main practical limitation in performing computation over encrypted data. We identify that multiplication can be eliminated when one of the operands of a multiplication is 0 or 1 in plaintext. In addition, consider the case of calculating $pow(x, n)$, where x is an encrypted number. The compiler can divide the computation into subproblems of size $n/2$ and call the subproblems recursively. Fig. 7 shows the program of computing $pow(2, 8)$, where 2 is private. The Harpoon program is transformed to HIR code, and is further generated to the TFHE program, where the function `unum_mul` multiplies two 64-bit encrypted numbers. The generated program only needs $O(\log n)$ multiplies. This optimization is simple, but has a dramatic impact on performance.

The effectiveness of the optimization is clearly demonstrated in Fig. 8, which shows the graphs of the generated circuits. The left (before optimization) is a depth-7 circuit with 7 multiply gates. The right (after optimization) is a depth-3 circuit with three multiply gates.

The generated graphs show an abstract model of execution cost where each operation is treated as atomic. However, the resource estimation framework can be specialized to particular backends by providing the corresponding models of execution cost (in terms of communication complexity, number of logic gates, etc.). These backend-specific resource estimates can be used to compare different optimization strategies and intelligently select the appropriate one based on the execution semantics of the targeted backend. In addition, as mentioned in Sec. 3.5, these specialized estimates even let us pick the most optimal backend to target.

5.2 Private Comparison

The private comparison is a major bottleneck in MPC protocols due to their inherent non-arithmetic structure [14]. Private comparison operators include <, ≤, >, ≥, == and ≠. One operator may be encoded by two or more other operators. However, the two expressions may have different costs. We identify some implementation heuristics that help us generate efficient programs.

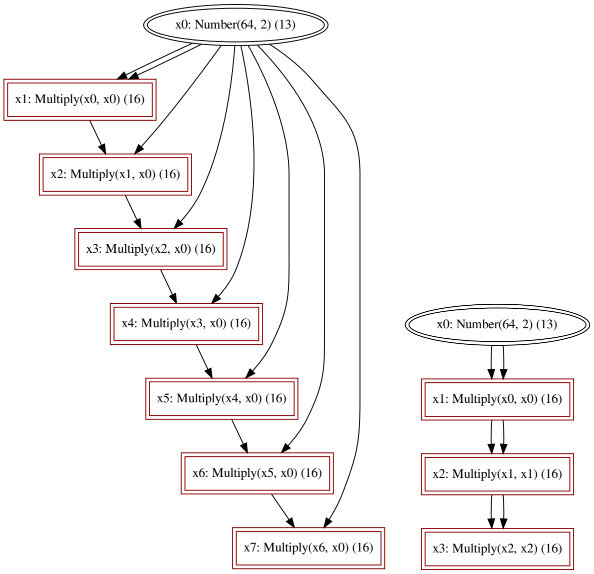
x0: Number(64, 2) (13)
x1: Multiply(x0, x0) (16)
x2: Multiply(x1, x0) (16)
x3: Multiply(x2, x0) (16)
x4: Multiply(x3, x0) (16)
x5: Multiply(x4, x0) (16)
x6: Multiply(x5, x0) (16)
x7: Multiply(x6, x0) (16)
x0: Number(64, 2) (13)
x1: Multiply(x0, x0) (16)
x2: Multiply(x1, x1) (16)
x3: Multiply(x2, x2) (16)

Figure 8. Graphs of computing $pow(2, 8)$: before (left) and after applying optimizations (right).

For example, the HoneyBadgerMPC library provides two comparison protocols: LessThan and Equality. They are used to express $a < b$ and $a == b$ on shared values, and return a secret shared value. Building an MPC compiler requires us to implement other operators in terms of these two. For example, a naive and intuitive implementation is to encode $a \geq b$ as $(b < a) + (a == b)$. An alternative way is to encode it as $1 - (b < a)$. Our abstract resource estimator generates one LEQ, one ADD and one EQUAL gate for the first encoding, and one SUB and one LEQ gate for the second encoding. In the HoneyBadgerMPC resource model, the costs of addition and subtraction are trivial since they require no communication, and the multiplication takes one round and one multicast to finish. The round complexity of comparison is seven times more than the cost of multiplication [36], the communication cost is even more expensive. Also, the cost of equality check is higher than the less than operation. Thus, we believe that the second encoding is better due to the reduced number of comparison. This demonstrates how we experiment optimizations guided by our resource estimators.

To verify the above observation, we perform a set of private comparison in a HoneyBadgerMPC program (on the same machine used in Sec. 6). Our tests execute 100 times

Table 1. Execution time of evaluating $a \geq b$ for 100 times, where a and b are randomly generated number ranging from 1 to 100.

Encoding $a \geq b$	Execution Time
$(b < a) + (a == b)$	0.23s
$1 - (b < a)$	0.10s

of the greater or equal comparison on two randomly generated numbers. Table 1 compares the running time of the two encodings.

6 Evaluation

This section presents three case studies to assess our framework focusing on Harpoon and HIR, optimizing scalar multiplication, and support for indexing arrays with secrets, respectively. For simplicity, the test program uses plaintext values instead of obtaining them at runtime. We conducted our experiments on a machine with 8 Intel Core i7 processors and 16 GB RAM that runs Ubuntu 18.04 LTS.

6.1 Case Study 1: Secure Auctions

Recall the discussion of the practical importance of secure auctions in Sec. 2. This experiment implements a second-price auction that is designed to give bidders confidence to bid their best price without overpaying. The bidder who submits the highest bid is awarded the item and pays the amount of the second-highest bid.

Fig. 9 shows the code snippets in Harpoon, where the elements in arrays `bidders` and `bid` denote bidder's identities and their bids. The implementation uses four variables (`fst`, `snd`, `ifst` and `isnd`) to store the values of the first and second highest bids and the identities of holders respectively. As shown in Fig. 9, writing the Harpoon implementation does not require developers to have cryptographic concerns or circuit building mindset. They can program functionally or imperatively, thanks to the expressiveness of Scala.

As mentioned in Sec. 2, our compiler could transform the imperative Harpoon program to a functional style one as `(bids zip bidders).map(..).reduce(..)`, which yields a circuit of logarithmic depth that allows efficient parallel computation.

We have generated SPDZ and HoneyBadgerMPC programs to realize secure auctions. For testing and development, the HoneyBadgerMPC program runs in a simulated network, and contains lines of code dealing with network connections and synchronizations. The Harpoon and HIR developers need not to have those concerns.

6.2 Case Study 2: Matrix-Vector Product

Secure matrix-vector multiplication is a core kernel in many real-world applications. For example, in the area of privacy-preserving machine learning, matrix-vector multiplication is one of the common building blocks of neural networks [47]. During the training and inference procedures, it is often the case that multiple parties combine their data where secure matrix-vector multiplication can be used to preserve privacy.

```
@sec var bidders = Array(0, 1 ..., n - 1)
@sec var bids = Array(b1, b2, ..., bn)
var ifst = bidders(0)     var isnd = bidders(1)
var fst = bids(0)         var snd = bids(1)
if (bids(0) < bids(1)) {
  ifst = bidders(1)   fst = bids(1)
} else {
  isnd = bidders(1)   snd = bids(1)
}
for (i <- 2 until bids.length) {
  if (fst < bids(i)) {
    isnd = ifst         snd = fst
    ifst = bidders(i)   fst = bids(i)
  } else if (snd < bids(i)) {
    isnd = bidders(i)   snd = bids(i)
  }
}
(ifst, snd)
```

Figure 9. Harpoon code snippet performing a second-price auction, where `b1, b2, ..., bn` are parameters passed to the method.

```
val rand = new scala.util.Random
val start = 1000
@sec(alice) val m = Array.fill(10)(
                start + rand.nextInt(start + 1))
val v = Array(1, 399, 1, 413, 1, 587, 1, 354, 1, 444)
m * v
```

Figure 10. Test program of Matrix-Vector Multiplication.

The case study performs a set of secure matrix-vector multiplication, where one party (the client) has an input matrix, and the other party (the server) has a vector. Fig. 10 shows the test program that randomly generates a $10 * N$ matrix (where $100 \leq N \leq 500$), and multiplies with a fixed vector $[1, 399, 1, 413, 1, 587, 1, 354, 1, 444]$. The test shows the effectiveness of our optimization discussed in Sec. 5.1.

Table 2 compares the running time of the generated HElib programs with and without optimizations. As N increases from 100 to 500, the speedups become more observable. We have provided median of absolute runtime before and after the optimization with 95% confidence.

6.3 Case Study 3: Merge Sort

MergeSort is a key computation component of various Secure MPC applications. For example, when multiple parties exchange messages anonymously, both the content and the metadata (*e.g.*, the length of the message) need to be protected. Secure sort is one of the core kernels used for such anonymous communications [1].

This case-study implements MergeSort in HIR, as it exposes the language features needed in writing secure computation. The implementation involves array indexing and conditional executions. Notably, an array lookup on a private index is not supported by most programming languages [23].

Table 2. Execution time (in seconds) of the HElib programs that perform multiplication of a matrix of $10 * N$ and a vector $[1, 399, 1, 413, 1, 587, 1, 354, 1, 444]$ before and after the optimization.

N	100	200	300	400	500
Before (median)	7.57	21.31	55.81	135	292
Error (confidence 95%)	0.51	0.44	0.72	1.45	2.05
After (median)	6.32	15.18	34.89	80	162
Error (confidence 95%)	0.05	0.34	0.23	0.34	0.49

```
1   val o1 = Owner()    val s = 0
2   var arr = NumArray(o1, 3, 1, 5, 2)  // input
3   val e = arr.length
4   def merge(o: Owner, arr1: NumArray, arr2: NumArray) = {
5     var res = NewNumArray(o, arr1.length + arr2.length)
6     var i = Num(o, 0)   var j = Num(o, 0)  var k = 0
7     while (k < res.length) {
8       val b1 = i < Num(o, arr1.length)
9       val b2 = j < Num(o, arr2.length)
10      val p = if (b1.not) arr2(j) else if (b2.not) arr1(i)
11              else if (arr1(i) <= arr2(j)) arr1(i) else arr2(j)
12      res = res.update(k, p)
13      // updating arr1 index
14      i = if (b1.not) i else if (b2.not) i + Num(o, 1) else
15          if (p == arr1(i)) i + Num(o, 1) else i
16      // updating arr2 index
17      j = if (b1.not) j + Num(o, 1) else if (b2.not) j
18          else if (p == arr2(j)) j + Num(o, 1) else j
19      k = k + 1
20    }
21    res
22  }
23  val r = recFuel(10);
24  val mergesort = r.rec[NumArray, Owner, Int, Int] {
25                  f => (a, o, i, j) => {
26    val mid = (j - i) / 2
27    if (mid == 0 || i >= j){ a }
28    else {
29      val left = a.slice(i, mid)  val right = a.slice(mid, j)
30      merge(o, f(left, o, 0, left.length),
31               f(right, o, 0, right.length))}
32    }
33  }
34  val res = mergesort(arr, o1, s, e)
35  output(res.eval(o1))
```

Figure 11. MergeSort implemented in HIR.

MergeSort recursively divides an input array into two halves and then merges the two sorted halves. Our implementation is shown in Fig. 11. In the function mergesort, the variable r (line 23) stores a recursion object initialized with the bound 10. The expression r.rec (line 24) is the construct for defining a bounded recursive function call. This allows one to explicitly specify the bound of the defining recursive function. The NumArray is the type for arrays that allow private indexing. The two parameters i and j are plaintext, which is important for unrolling the recursive function at compile time. The function slice(i, j) returns a subarray from the ith element until the jth element, where i and j are plaintext integers. The if-statement (lines 27 to 31) is the standard one as its condition depends on a plain text value. The function merge is used for merging two halves. All the if-constructs appearing in this function are oblivious as their conditions depend on ciphertext values. The loop (line 7) is bounded as the length of an array is known at compile time.

7 Related Work

There have been many MPC frameworks proposed in recent years and several of them are already integrated into HACCLE. We list the prominent MPC frameworks as follows.

SCALE-MAMBA [29] is an existing MPC framework that is closest to HACCLE. We utilize it as one of our cryptographic backends to implement secret sharing and FHE based protocols. It is a combination of a compiler and a run-time environment where optimizations can be performed at a lower level. Compared with SCALE-MAMBA, HACCLE provides staging driven by type systems, estimates resource consumption, and focuses on optimization at a higher level.

HoneybadgerMPC [32] is another backend of HACCLE that supports secret-sharing based protocols. The uniqueness of HoneybadgerMPC is the combination of a robust online phase and an optimal non-robust offline phase. It provides fairness guarantees even in the asynchronous network setting and also preserves efficiency to make MPC programs practical to run.

As privacy preserving machine learning becomes more and more popular, many frameworks have been developed specifically for this use case, such as ABY [17], ABY3 [34], CHET [16], EzPC [9], CrypTFlow [30] and SecureNN [47]. These frameworks are highly optimized for machine learning and are designed for two-party or three-party settings. We choose not to include them due to our desire to support an arbitrary number of parties. There are also many other MPC frameworks such as Viff [45], Jiff [43], MPyC [4] and PICCO [52]. Theoretically, any framework can be embedded as a backend in HACCLE even though not all of them are integrated at the moment.

8 Conclusion

Secure MPC-based applications play a crucial role in solving many important practical problems such as in high-value procurement. But developing performant MPC-based applications from scratch is a notoriously difficult task as it requires expertise ranging from cryptography to circuit optimization. Therefore software developers need a compiler toolchain for developing MPC-based applications. As a solution to this problem, we have introduced the HACCLE toolchain, a multi-stage compiler for optimized circuit generation. We believe that the HACCLE toolchain offers a compelling approach to the design and implementation of Secure MPC applications, using metaprogramming techniques.

Acknowledgments

We thank the anonymous reviewers for their helpful suggestions and comments. This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), contract #2019-19020700004. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

[1] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. 2017. MCMix: Anonymous Messaging via Secure Multiparty Computation. In *USENIX Security Symposium*. USENIX Association, 1217–1234. http://eprint.iacr.org/2017/778

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[3] Yuyan Bao, Kirshanthan Sundararajah, Raghav Malik, Qianchuan Ye, Christopher Wagner, Nouraldin Jaber, Fei Wang, Mohammad Hassan Ameri, Donghang Lu, Alexander Seto, Benjamin Delaware, Roopsha Samanta, Aniket Kate, Christina Garman, Jeremiah Blocki, Pierre-David Letourneau, Benoît Meister, Jonathan Springer, Tiark Rompf, and Milind Kulkarni. 2020. HACCLE: Metaprogramming for Secure Multi-Party Computation - Extended Version. *CoRR* abs/2009.01489 (2020). https://arxiv.org/abs/2009.01489

[4] Barry Schoenmakers. 2020. MPyC: Secure multiparty computation in Python. https://github.com/lschoe/mpyc

[5] Peter Bogetoft, Ivan Damgård, Thomas P. Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. 2006. A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In *Financial Cryptography (Lecture Notes in Computer Science, Vol. 4107)*. Springer, 142–147. https://doi.org/10.1007/11889663_10

[6] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *CRYPTO (Lecture Notes in Computer Science, Vol. 7417)*. Springer, 868–886. https://doi.org/10.1007/978-3-642-32009-5_50

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. *Electron. Colloquium Comput. Complex.* (2011), 111. https://eccc.weizmann.ac.il/report/2011/111

[8] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. 2010. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In *USENIX Security Symposium*. USENIX Association, 223–240. http://www.usenix.org/events/sec10/tech/full_papers/Burkhart.pdf

[9] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *EuroS&P*. IEEE, 496–511. https://doi.org/10.1109/EuroSP.2019.00043

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT (1) (Lecture Notes in Computer Science, Vol. 10624)*. Springer, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *ASIACRYPT (1) (Lecture Notes in Computer Science, Vol. 10031)*. 3–33. https://doi.org/10.1007/978-3-662-53887-6_1

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2017. Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE. In *ASIACRYPT (1) (Lecture Notes in Computer Science, Vol. 10624)*. Springer, 377–408. https://doi.org/10.1007/978-3-319-70694-8_14

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. August 2016. TFHE: Fast Fully Homomorphic Encryption Library. https://tfhe.github.io/tfhe/

[14] Geoffroy Couteau. 2016. Efficient Secure Comparison Protocols. *IACR Cryptol. ePrint Arch.* (2016), 544. http://eprint.iacr.org/2016/544

[15] 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 *ESORICS (Lecture Notes in Computer Science, Vol. 8134)*. Springer, 1–18. https://doi.org/10.1007/978-3-642-40203-6_1

[16] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *PLDI*. ACM, 142–156. https://doi.org/10.1145/3314221.3314628

[17] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*. The Internet Society. https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation

[18] Jack Doerner, David Evans, and Abhi Shelat. 2016. Secure Stable Matching at Scale. In *CCS*. ACM, 1602–1613. https://doi.org/10.1145/2976749.2978373

[19] Grégory M. Essertel, Guannan Wei, and Tiark Rompf. 2019. Precise reasoning with structured time, structured heaps, and collective operations. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 157:1–157:30. https://doi.org/10.1145/3360583

[20] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.* 2, 2-3 (2018), 70–246. https://doi.org/10.1561/3300000019

[21] Nicolas Gama, Malika Izabachène, Phong Q. Nguyen, and Xiang Xie. 2016. Structural Lattice Reduction: Generalized Worst-Case to Average-Case Reductions and Homomorphic Cryptosystems. In *EUROCRYPT (2) (Lecture Notes in Computer Science, Vol. 9666)*. Springer, 528–558. https://doi.org/10.1007/978-3-662-49896-5_19

[22] Trinabh Gupta, Henrique Fingler, Lorenzo Alvisi, and Michael Walfish. 2017. Pretzel: Email encryption and provider-supplied functions are compatible. In *SIGCOMM*. ACM, 169–182. https://doi.org/10.1145/3098822.3098835

[23] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. Sok: General purpose compilers for secure multiparty computation. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1220–1237. https://doi.org/10.1109/SP.2019.00028

[24] Markus Hinkelmann, Andreas Jakoby, Nina Moebius, Tiark Rompf, and Peer Stechert. 2011. A cryptographically *t*-private auction system. *Concurr. Comput. Pract. Exp.* 23, 12 (2011), 1399–1413.

[25] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. 2012. Secure two-party computations in ANSI C. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 772–783. https://doi.org/10.1145/2382196.2382278

[26] Karthik A Jagadeesh, David J Wu, Johannes A Birgmeier, Dan Boneh, and Gill Bejerano. 2017. Deriving genomic diagnoses without revealing patient genomes. *Science* (2017).

[27] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. *SIGPLAN Not.* 53, 4 (June 2018), 296–311. https://doi.org/10.1145/3296979.3192379

[28] Benjamin Kreuter. 2017. Secure MPC at Google. Real World Crypto.

[29] KU Leuven. 2019. SCALE-MAMBA Software. https://homes.esat.kuleuven.be/~nsmart/SCALE/.

[30] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow: Secure TensorFlow Inference. In *IEEE Symposium on Security and Privacy*. IEEE, 336–353. https://doi.org/10.1109/SP40000.2020.00092

[31] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376. https://doi.org/10.1109/SP.2015.29

[32] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 887–903. https://doi.org/10.1145/3319535.3354238

[33] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security Symposium*. USENIX, 287–302. http://www.usenix.org/publications/library/proceedings/sec04/tech/malkhi.html

[34] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*. ACM, 35–52. https://doi.org/10.1145/3243734.3243760

[35] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 655–670. https://doi.org/10.1109/SP.2014.48

[36] Tord Ingolf Reistad and Tomas Toft. 2007. Secret sharing comparison by transformation and rotation. In *International Conference on Information Theoretic Security*. Springer, 169–180. https://doi.org/10.1007/978-3-642-10230-1_14

[37] Tiark Rompf and Kevin J Brown. 2017. Functional parallels of sequential imperatives (short paper). In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 83–88. https://doi.org/10.1145/3018882.3018891

[38] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 127–136. https://doi.org/10.1145/1868294.1868314

[39] Victor Shoup Shai Halevi. April 2013. HElib: Design and Implementation of a Homomorophic-Encryption Library. https://github.com/shaih/HElib

[40] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[41] Walid Taha. 2003. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation (Lecture Notes in Computer Science, Vol. 3016)*. Springer, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

[42] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

[43] Multiparty.org Development Team. 2020. JavaScript implementation of federated functionalities. https://github.com/multiparty/jiff

[44] The PALISADE team. 2021. *PALISADE, homomorphic encryption softare library*. https://palisade-crypto.org/

[45] The VIFF team. 2021. *VIFF, the virtual ideal functionality framework*. http://viff.dk/

[46] William Vickrey. 1961. Counterspeculation, Auctions, and Ccompetitive Sealed Tenders. *The Journal of Finance* 16, 1 (1961), 8–37. https://doi.org/10.1111/j.1540-6261.1961.tb02789.x

[47] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 26–49. https://doi.org/10.2478/popets-2019-0035

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

[49] Xiao Shaun Wang, Yan Huang, Yongan Zhao, Haixu Tang, XiaoFeng Wang, and Diyue Bu. 2015. Efficient Genome-Wide, Privacy-Preserving Similar Patient Query based on Private Edit Distance. In *CCS*. ACM, 492–503. https://doi.org/10.1145/2810103.2813725

[50] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *FOCS*. IEEE Computer Society, 160–164. https://doi.org/10.1109/SFCS.1982.38

[51] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.* (2015), 1153. http://eprint.iacr.org/2015/1153

[52] 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*. 813–826. https://doi.org/10.1145/2508859.2516752