An Array Intermediate Language for Mixed Cryptography

Vivian Ding
Cornell University
Ithaca, NY, USA
vyd2@cornell.edu

Coşku Acay
Cornell University
Ithaca, NY, USA
cacay@cs.cornell.edu

Andrew C. Myers
Cornell University
Ithaca, NY, USA
andru@cs.cornell.edu

1 INTRODUCTION

Advanced cryptographic mechanisms such as secure multiparty computation (MPC) [Yao82], zero-knowledge proofs (ZKP) [GMR89], and fully homomorphic encryption (FHE) [Gen09] all expect programs represented as fixed-sized circuits. Since it is painful to program circuits directly, cryptographic compilers [MNPS04, IMZ19, ACK+19, RHH14, LWN+15, PHGR13, CFH+15, WSR+15, ARG+21, LSI+23, DSC+19, DKS+20, CDA+21, VJHH22, MSK23, JVC18] alleviate the burden on programmers by translating high-level code to low-level circuits. However, many existing compilers fail to take full advantage of the underlying cryptographic libraries to speed up execution, as neither the source representation nor the circuit representation are suitable for optimizations such as vectorization.

Prior work on cryptographic intermediate representations (IRs), including that focusing on vectorization for cryptographic mechanisms [LSI+23, OBW22], targets programs whose entire behavior can be implemented as a single cryptographic circuit. This approach therefore cannot support interactive programs in which user input arrives during execution; nor does it support the compilation of programs mixing multiple cryptographic back ends and local computation, which is useful for performance.

We introduce AIRduct, a new array-based intermediate representation designed to support generating efficient code for interactive programs employing multiple cryptographic mechanisms. AIRduct is intended as an IR for the Viaduct compiler [ARG+21], which can synthesize secure, distributed programs with an extensible suite of cryptography. Therefore, AIRduct supports an extensible variety of cryptographic mechanisms, including MPC and ZKP. It is the job of the Viaduct compiler to select cryptographic protocols that make the compiled program secure, guided by information-flow annotations. In this paper, we assume this choice has been made correctly, and focus on the IR. A proof that Viaduct generates secure target code in the sense of UC simulation [Can01], assuming that cryptographic mechanisms are described faithfully by the information-flow annotations, is provided elsewhere [AGRM23].

2 OVERVIEW OF AIRDUCT

Large-scale applications may require multiple cryptographic mechanisms at different points of a program, as well as reactive functionality or running computations indefinitely. AIRduct represents code employing multiple cryptographic mechanisms as structured control flow over calls to circuit functions. To demonstrate how our approach enables these programs, we use a program for interactive biometric matching as a running example. In listing 1, the circuit function `biometric` describes a computation to be performed on `MPC`, which represents a secure multiparty computation protocol

```
1  host Client, Server
2
3  circuit fun <n, d> biometric@MPC(
4    database: int[n, d],
5    sample: int[d]) → min_dist: int[] {
6      val dists[i < n] = reduce(::+, 0) { j < d →
7          (database[i, j] - sample[j]) ** 2
8      }
9      val res[] = reduce(::min, int.MAX) { i < n →
10         dists[i]
11     }
12     return res
13 }
14
15 fun main() {
16     val N@Replication(Client, Server) = 1000
17     val D@Replication(Client, Server) = 20
18     val database@Local(Server) =
19         Server.input<int[N, D]>()
20     while (true) {
21         val sample@Local(Client) =
22             Client.input<int[D]>()
23         val result@Local(Client) =
24             biometric<N, D>(database, sample)
25         Client.output(result)
26     }
27 }
```

Listing 1: An interactive biometric matching service.

between `Server` and `Client`. It matches a `d`-tuple sample against a database of `n` points and reports the squared Euclidean distance between the sample and the closest database entry. The result is a zero-dimensional array (a scalar), as indicated by the syntax `int[]`.

The function `main` performs an infinite loop of inputs, outputs, and calls to the circuit function `biometric`. Results of function calls are bound to variables, which are associated with *storage formats* describing how the variables are stored across hosts. In the example, values specifying array sizes are stored using a replication protocol, as both parties must have access to array sizes. The input samples and results of calling `biometric` are stored locally by `Client`.

3 SYNTAX

Figure 1 gives the syntax of AIRduct. Circuit functions are straight-line blocks of computations performed on a single cryptographic protocol, parameterized by array sizes. Computations in circuit functions are expressed in the form of assignments to multidimensional arrays. The expressions used in these assignments include arithmetic and logical operations, indexing into arrays, references to named values (including size parameters), and bulk operations such as **reduce**. While each circuit function must be associated with

FCS 2023, July 9, 2023, Dubrovnik, Croatia
2023.

Hosts $h \in \mathbb{H}$ Computation Protocol $p \in \mathbb{P}$
Variables $x \in \mathbb{X}$ Storage Protocol $q \in \mathbb{Q}$

Values	v	$\in$	$\mathbb{V}$
Types	τ	$\in$	$\mathbb{T}$
Function Names	f	$\in$	$\mathbb{F}$
Atomic Expr.	t	::=	$v \mid x$
Index Bounds	b	::=	$x < t$
Scalar Expr.	e	::=	$x[\overline{t}] \mid op(e_1, e_2)$
		$\mid$	$\mathbf{reduce}(op, e)\{b \rightarrow e\}$
Commands	m	::=	$t \mid f\langle\overline{t}\rangle(\overline{t})$
		$\mid$	$h.\mathbf{input}\langle\tau\rangle() \mid h\mathbf{output}(t)$
Circuit Statements	c	::=	$\mathbf{val}\ x[\overline{b}] = e$
Statements	s	::=	$\mathbf{val}\ x@q = m$
		$\mid$	$\mathbf{if}\ t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$
Parameters	r	::=	$x : \tau$
Top-level Decl.	d	::=	
	$\mid$	$\mathbf{circuit\ fun}\ \langle\overline{x}\rangle\ f@p(\overline{r}) \rightarrow \overline{r}\ \{\overline{c}; \mathbf{return}\ \overline{x}\}$	
	$\mid$	$\mathbf{fun}\ \langle\overline{x}\rangle\ f(\overline{r}) \rightarrow \overline{r}\ \{\overline{s}; \mathbf{return}\ \overline{x}\}$	

Figure 1: Syntax of AIRduct.

one protocol, a program may employ multiple protocols across circuit functions. Non-circuit functions contain control-flow statements (conditionals) and let-bindings, which bind constants, inputs, outputs, and the results of circuit or function calls to named values. We elide looping constructs in the formal language since they can be recovered through recursion. No computation can occur in non-circuit functions.

4 ARRAY-BASED COMPUTATION

It is a standard restriction in cryptographic mechanisms such as MPC that bounds must be known at the time of circuit generation. Cryptographic circuits require all loops to be unrolled, conditional statements to be translated into "muxes", and functions to be inlined [MNPS04]. However, this process blows up the program size and erases all structure in the resulting circuit representation, making it difficult and expensive to perform optimizations.

Array programs are conditional and loop-free, and are therefore a useful representation for translation to low-level circuits, unlike high-level languages. Many loops in practice can be expressed with array operations, so programs do not grow unreasonably in size. Array programs also preserve structure and natively capture bulk operations, so they are easy to optimize and parallelize, unlike circuits. As vectorization is a crucial optimization for efficiency [LSI+23, BDST22, CDA+21, VJHH22, MSK23], this array-centric representation may present significant speedup in generated code.

As such, computations in AIRduct are expressed solely using arrays. Circuit functions in AIRduct are parameterized over sizes to support protocols which require bounds to be known at generation time.

5 CONTROL FLOW

Additionally, AIRduct separates control flow from cryptographic execution by restricting all computation to pure circuit blocks. Inputs, outputs, calls to circuits, and other control-flow statements occur only in non-circuit functions. This representation prevents arbitrary control flow inside of circuits, without restricting the overall program to straight-line computations. It also enables reactivity: programs can exit from circuit functions to perform input and outputs before resuming computation.

Thus, in the biometric example, while the size of the circuit is bounded, the interactive loop between the client and server can continue indefinitely.

6 PROTOCOL MIXING

Intermediate storage may be needed if values computed by a circuit function are used in other circuits. For some cryptographic implementations such as ABY [DSZ15], circuits are destroyed upon evaluation. In order to support intermediate results, values must be explicitly exported and imported as secret shares.

In AIRduct, cryptographic protocols may be used for computation or intermediate storage. For instance, commitment schemes and secret sharing are storage protocols, as they specify a format for storing data. ZKP and MPC are computation protocols, and replication is both a storage and a computation protocol. Each protocol is associated with data formats for storing or computing values. To execute circuit functions, data must be imported and exported between these formats.

In the biometric example, the call to `biometric` is associated with two imports (`database` and `sample`, imported from local cleartext formats to the format for computation with MPC) and one export (`result`, exported from the MPC computation format to the client's local storage).

Protocol back ends define how both to *import* values to the format used for cryptographic execution and to *export* data to intermediate storage. For instance, to export local values to the replicated format (in which data is replicated on all hosts involved with the protocol), hosts involved in replication must perform equivocation checks to ensure that the value they receive is the same as other hosts in the replication protocol. Libraries for MPC such as ABY [DSZ15] allow for execution of circuits in multiple schemes (arithmetic sharing, boolean sharing, and Yao's garbled circuits), and ABY supports transferring data between them.

Thus, defining communication between protocols in terms of transfers between storage and execution formats provides a useful abstraction for interactive computation that mixes cryptography.

7 GENERATION OF AIRDUCT

Circuit functions provide cryptographic back ends with large, contiguous blocks of straight-line computation as opposed to individual instructions. Working with blocks exposes parallelism opportunities and enables more advanced optimizations such as algebraic rewrites to simplify computations [WNW+21]. Additionally, large blocks amortize the cost of interfacing with cryptographic frameworks: building, evaluating, and destroying circuits, and importing and exporting data. Therefore, generating an IR which packs as

much computation as possible into each circuit function is advantageous for efficiency.

We propose to automatically *split* the input program into (large) circuit functions. The input to splitting is an array program which freely mixes computation and control flow, and where each statement is annotated with the protocol executing that statement.[1] The splitting procedure attempts to group together statements on the same protocol by reordering them. The goal of reordering is to maximize block size and to minimize data dependence between blocks (which becomes communication in the form of imports and exports in the generated IR).

The compiler cannot freely reorder statements: in addition to being restricted by data dependencies, some reorderings violate security. The compiler cannot move an **`output`** statement before an **`input`** statement, and it cannot move a statement that reveals information before one that commits to data. For example, if the source program states `Client` guesses a number and then `Server` reveals the secret number it picked, the compiler should not switch the order of these statements. These constraints precisely identify when it is safe to reorder [AGRM23].

8 CONCLUSION

We present a new intermediate representation for compilers that generate code for advanced cryptographic libraries. Our IR supports interactive programs by making explicit the boundary between control flow and cryptographic computation. It allows mixing local computation with multiple different cryptographic mechanisms by distinguishing storage formats from computation protocols. Finally, it facilitates vectorization and other optimizations by partitioning source programs into large contiguous blocks of array programs. Fully integrating the new IR into Viaduct is a work in progress. We are working on implementing splitting, and taking full advantage of vectorization. We have integrated replication and MPC (through ABY [DSZ15]), and are hoping to integrate ZKP and fully homomorphic encryption (FHE).

REFERENCES

[ACK+19] Abdelrahaman Aly, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. *SCALE–MAMBA v1.6 : Documentation*, 2019.

[AGRM23] Coşku Acay, Joshua Gancher, Rolph Recto, and Andrew Myers. Secure synthesis of distributed cryptographic applications. In submission, 2023.

[ARG+21] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew Myers, and Elaine Shi. Viaduct: An extensible, optimizing compiler for secure distributed programs. In *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 740–755. ACM, June 2021.

[BDST22] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. Motion – a framework for mixed-protocol multi-party computation. *ACM Trans. Priv. Secur.*, 25(2), 2022.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Symposium on Foundations of Computer Science (FOCS)*, pages 136–145. IEEE Computer Society, 2001.

[CDA+21] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 375–389, 2021.

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Symp. on Security and Privacy*, pages 253–270. IEEE, 2015.

[CLM+07] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *21st ACM Symp. on Operating System Principles (SOSP)*, pages 31–44, October 2007.

[DKS+20] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In Alastair F. Donaldson and Emina Torlak, editors, *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 546–561. ACM, 2020.

[DSC+19] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In Kathryn S. McKinley and Kathleen Fisher, editors, *40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 142–156. ACM, 2019.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security Symp.* The Internet Society, 2015.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *41st ACM Symp. on Theory of Computing*, pages 169–178, 2009.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[IMZ19] Muhammad Ishaq, Ana Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *26th ACM Conf. on Computer and Communications Security (CCS)*, pages 1539–1556. ACM, 2019.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1651–1669. USENIX Association, 2018.

[LSI+23] Benjamin Levy, Ben Sherman, Muhammad Ishaq, Lindsey Kennard, Ana L. Milanova, and Vassilis Zikas. Compilation and backend-independent vectorization for multi-party computation. *IACR Cryptol. ePrint Arch.*, page 89, 2023.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *25th ACM Symp. on Operating System Principles (SOSP)*, pages 359–376. IEEE, 2015.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - a secure two-party computation system. In *13th Usenix Security Symposium*, pages 287–302, August 2004.

[MSK23] Raghav Malik, Kabir Sheth, and Milind Kulkarni. Coyote: A compiler for vectorizing encrypted arithmetic circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 118–133, 2023.

[OBW22] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *IEEE Symp. on Security and Privacy*, pages 2248–2266, 2022.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symp. on Security and Privacy*, pages 238–252. IEEE, 2013.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symp. on Security and Privacy*, pages 655–670, May 2014.

[VJHH22] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. HECO: automatic code optimizations for efficient fully homomorphic encryption. *CoRR*, abs/2202.01649, 2022.

[WNW+21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[WSR+15] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symp.* The Internet Society, 2015.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *23rd annual IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[ZCMZ03] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.

[1] Protocol annotations can be provided by the programmer or inferred automatically through information-flow analysis [ARG+21, ZCMZ03, CLM+07].