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 [MNP04, IMZ19, ACK+19, RHH14, LWN+15, PHGR13, CFH+15, WSR+15, ARG+21, LSI+23, DSC+19, DKS+20, CDA+21, VJJH22, MSK23, JVC+18] 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 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 $\text{MPC}$, which represents a secure multiparty computation protocol 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 $\text{int}[]$.

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.

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 $\text{MPC}$, which represents a secure multiparty computation protocol 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 $\text{int}[]$.

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...
Array programs also preserve structure and natively capture bulk support protocols which require bounds to be known at generation representation may present significant speedup in generated code. Array operations, so they are easy to optimize and parallelize, unlike circuit operations, so programs do not grow unreasonably in size. High-level languages. Many loops in practice can be expressed with a useful representation for translation to low-level circuits, unlike 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 equivalence 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 computation with multiple different cryptographic mechanisms by distinguishing storage formats from computation protocols. Finally, we generate code for advanced cryptographic libraries. Our IR supports splitting, and taking full advantage of vectorization for multi-party computation. [IACR Cryptol. ePrint Arch., page 89, 2023.

REFERENCES


---

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