Research Statement: Raghav Malik

In a world increasingly concerned about data privacy, the availability of privacy-preserving computation is more important than ever. Whether it is to protect intellectual property, as in the case of secure machine learning, or to comply with governmental data privacy regulations like HIPAA or GDPR, we often need to write applications that can perform computations without learning anything about their inputs. A popular approach to writing privacy-preserving software is Multiparty computation, or MPC. Multiparty computation (MPC) refers to a class of cryptographic protocols that allow mutually distrusting parties to jointly compute functions over private inputs, without the need to share their inputs with each other. While these protocols are in theory powerful enough to evaluate any (bounded, terminating) function, in practice, actually writing programs that use MPC has a number of drawbacks. The cryptographic overhead is often so high that nontrivial MPC programs can be prohibitively expensive to evaluate. Furthermore, the MPC programming model can be highly counterintuitive, making it very difficult for non-experts to write efficient MPC programs. My research aims to *make it easier for non-experts to write privacy-preserving software* by *building compilers for MPC*.

Compilers naturally suggest themselves as solutions to the drawbacks of MPC, promising to obviate the need for a deep understanding of the cryptosystem by automatically generating efficient MPC code from high level programs. Unfortunately, most modern MPC compilers take essentially the same approach that compilers did in the 80s, mechanically translating insecure operations into their MPC equivalents, and doing very little optimization on top of this. Of course, traditional compilers have come a long way since then, and optimizations like smart instruction selection and vectorization are now considered standard practice. However, transferring these techniques into the MPC space can be nontrivial. Vectorizing compilers, for example, typically target vector instruction sets in CPUs, and therefore bake in certain assumptions about the cost model of these instructions. A naïve attempt at applying traditional (SLP-style) vectorization to MPC fails for this exact reason: while some MPC schemes do support a limited form of vectorization called *ciphertext-packing*, their programming model is vastly different from that of CPU vector architectures, and the lack of an easy vector selection/permutation instruction means that SLP-vectorized code can easily perform *worse* than its scalar counterpart. The takeaway, then, is that just as a good compiler engineer needs to have a deep understanding of the architecture being targeted, *writing an MPC compiler requires a deep understanding of the semantics and cost model of the target MPC scheme!*

Research Approach

My research focuses on bringing this expertise to the world of MPC compilers, and using it to build compilers that are *MPC-aware*; that is, they can leverage knowledge about the target scheme to perform interesting and useful optimizations beyond simply translating programs into MPC operations. A good example of my applying this approach is Coyote [Malik et al., 2023], a compiler I developed for applying SLP-style vectorization to programs that target BFV, a particular implementation of an MPC scheme called Fully Homomorphic Encryption (FHE). In Coyote, I note how important the trade-off between vector scheduling and data layout is in the context of MPC vectorizers: a too-aggressive vector schedule can result in data layouts that require prohibitively expensive operations to move inputs to the correct lane. Unlike traditional SLP vectorizers, Coyote runs the searches for a schedule and data layout *together* rather than separately, and uses results from one search to guide the other, thus ensuring that the final generated schedule can be vectorized *without* incurring prohibitively expensive data movement operations. It first constrains the data layout by choosing a set of operations to force onto the same lane, and then uses a constrained form of SLP to find a vector schedule that satisfies the layout constraints. Finally, it uses the vector schedule to provide feedback about which operations to group together, and repeats the cycle. I show that Coyote can effectively vectorize arbitrary code while avoiding too much data-movement overhead, thus enabling it to outperform traditional SLP-style approaches. Additionally, I demonstrate that Coyote's techniques can often generate schedules that match the best expert-written implementations for some kernels.

Compiler-Cryptosystem Codesign

By not treating hardware as a fixed constant, compiler researchers can often go further than just optimizing programs for target architectures, and instead do *hardware-software codesign*. Similarly, instead of thinking of an MPC protocol as a fixed architecture to target, we can push further into *compiler-cryptosystem codesign*, in which we tweak the *protocol itself* to provide better abstractions for our compilers to target. My ongoing work on COATL demonstrates how powerful this approach can be. COATL is a compiler that targets another FHE scheme called CGGI. The usual programming model for CGGI exposes a number of "boolean gates" that can be evaluated securely; thus, most compilers that target CGGI essentially work by compiling programs into boolean circuits. In COATL, however, I notice that the boolean gates provided by the scheme are a mere abstraction over the actual operations implemented in the protocol: the ability to take linear combinations of ciphertexts, and the ability to securely index into lookup tables. I use these operations to develop the *arithmetic lookup table*, a new CGGI abstraction that strictly generalizes the old boolean gates. I then show that a compiler that targets these arithmetic lookup tables can produce circuits that are much smaller and more efficient than their boolean circuit counterparts.

Other FHE Compilers

Other work I have done in this space includes COPSE [Malik et al., 2021], a compiler for vectorizing decision tree inference, and COIL, an ongoing project that proposes a novel IR for representing FHE programs that contain branching control flow.

COPSE Running inference on decision trees is a challenging problem to solve in FHE: The usual (plaintext) inference algorithm involves traversing the tree by evaluating the condition at the root and then using that to decide which subtree to recurse on, but these control-flow dependences cannot be directly encoded in the FHE programming model without leaking information about the condition via the subtree that was chosen. Prior work solves this problem by transforming the tree into a polynomial that encodes the control-flow as data flow by evaluating every possible branch; however, the resulting polynomial is often too large and unwieldy to efficiently evaluate, and the complicated data dependences make it difficult to vectorize. In COPSE, I recognize that the unique semantics of FHE mean that every condition needs to be evaluated, allowing us to break the data dependences while preserving the semantics of the program. I develop a decision tree representation that encodes both the branching structure and the actual conditions as a series of matrices by breaking data dependences, and show how easily vectorizable matrix operations can be used to perform inference over this new representation. In the evaluation, I demonstrate how using the correct ("FHE-aware") set of abstractions for the problem can accelerate inference by up to an order of magnitude.

COIL In COIL, I further tackle the problem of compiling secure control flow. The usual strategy that FHE compilers use for dealing with branching control flow is to generate code that "multiplexes" by executing all branches and then using the branching condition to select between the possible results. I argue that the multiplexing strategy can give up a lot of optimization opportunities, since a multiplexed result inherently "forgets" which branch it came from. I propose the idea of *path forests*, a novel IR which keeps track of this information, allowing the compiler to perform a number of interesting peephole optimizations like path-dependant constant propagation, as well as more aggressive dead code elimination by identifying and pruning correlated branches. I also draw a connection between the path forest IR and the vectorizable representation developed in COPSE, and demonstrate how the COIL strategy can be adapted to effectively vectorize these branching programs.

I show that the optimizations enabled by the path forest IR can speed up generated code by several orders of magnitude, and find that COIL can automatically discover expert-designed implementations of several common kernels.

Future Research

While my work represents an important step forward in the world of MPC compilation, it still barely scratches the surface of what is possible. My vision for this space is to build MPC compilers that codify and automatically apply the decades of folklore tricks that MPC experts have used for manually writing efficient MPC programs. I plan to realize this vision by (1) Finding ways to generalize these tricks into principled techniques that MPC compilers can implement, as I did in Coyote and COIL, and (2) Developing novel abstractions at both the compiler and protocol level to make MPC programs easier to optimize, as I did in COPSE and COATL.

Short Term

There is a lot of low-hanging fruit in the space of FHE optimizations that I want to address in the short-term.

Coyote While Coyote's unique co-optimization strategy produces promising results, the expensive synthesis procedure means that it can fail to synthesize schedules for larger kernels. I want to investigate ways to augment Coyote's synthesis procedure with structural information about the program its vectorizing to shrink the search space and enable vectorizing much larger kernels. In particular, I note that many common kernels come equipped with natural "splitting points". I expect that a vectorization strategy that can break these kernels into smaller subprograms, vectorize each subprogram independently, and then intelligently recombine the schedules, can scale up to effectively vectorizing much larger kernels.

COIL Although the path-forest IR I proposed in COIL enables some very powerful optimizations in the presence of branching control flow, it is not well suited to representing programs where the branches are more naturally encoded as data flow branches (i.e. "muxes"). I want to develop extensions to the path forest IR that allow it to encode these programs as well, and extend the COIL compiler to automatically get "best of both worlds", using the path forest IR where it makes sense, and falling back to a more traditional IR when there aren't as many path-dependent optimizations available.

Scheme Switching Boura et al. [2018] and jie Lu et al. [2020] both propose techniques for switching ciphertexts between FHE schemes that offer different trade-offs. I want to investigate techniques for statically partitioning programs so that different subcomputations can run in different schemes.

Medium Term

In the more medium-term, I want to explore ways to generalize the techniques I have developed to apply to a broader class of MPC protocols, instead of being tailored for a specific scheme. For example, programmable bootstrapping, a feature which enables performing interesting computations during common ciphertext maintanence operations ("bootstrapping"), is currently largely unique to the programming model of the CGGI scheme (in particular, this programmability is what COATL exploits). However, recent work such as Kim et al. [2024] shows how to enchance bootstrap operations in other schemes like BFV. How can we generalize the CGGI programming model to allow for "hiding" more interesting computation in, e.g., BFV bootstrapping? Going beyond that, while most of my results so far are specific to particular FHE schemes, what would it take to extend these results to other MPC protocols? Better yet, can we fit these techniques into a more general (i.e., protocol-agnostic) framework for analyzing and optimizing MPC programs, and use this to automatically extract more protocol-specific optimizations? In HACCLE, we took some first steps in realizing this vision by building a staging compiler that enables both protocol-agnostic optimizations, and optimizations that can only be done once a protocol is chosen. While HACCLE established a framework for doing this kind of compilation, it did so in the naïve style discussed earlier. Bringing MPC-aware compilation techniques to a framework like HACCLE would be a great launching pad for new projects.

References

C. Boura, N. Gama, M. Georgieva, and D. Jetchev. CHIMERA: Combining ring-LWE-based fully homomorphic encryption schemes. Cryptology ePrint Archive, Paper 2018/758, 2018. URL https://eprint.iacr.org/2018/758.

W. jie Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. Cryptology ePrint Archive, Paper 2020/1606, 2020. URL https://eprint.iacr.org/2020/1606.

J. Kim, J. Seo, and Y. Song. Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS. Cryptology ePrint Archive, Paper 2024/109, 2024. URL https://eprint.iacr.org/2024/109.

R. Malik, V. Singhal, B. Gottfried, and M. Kulkarni. Vectorized secure evaluation of decision forests. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1049–1063, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454094. URL https://doi.org/10.1145/3453483.3454094.

R. Malik, K. Sheth, and M. 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*, ASPLOS 2023, page 118–133, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582057. URL https://doi.org/10.1145/3582016.3582057.