An Embedded Domain-Specific Language for Using One-Hot Vectors and Binary Matrices in Secure Computation Protocols

Andrei Lapets

Reity
Boston, MA
al@reity.org

Abstract

The use of secure computation protocols within production software systems and applications is complicated by the fact that such protocols sometimes rely upon – or are most compatible with – unusual or restricted models of computation. We employ the features of a contemporary and widely used programming language to create an embedded domain-specific language for working with user-defined functions as binary matrices that operate on one-hot vectors. At least when working with small finite domains, this allows programmers to overcome the restrictions of more simple secure computation protocols that support only linear operations (such as addition and scalar multiplication) on private inputs. Notably, programmers are able to define their own input and output domains, to use all available host language features and libraries to define functions that operate on these domains, and to translate inputs, outputs, and functions between their usual host language representations and their one-hot vector or binary matrix forms. Furthermore, these features compose in a straightforward way with simple secure computation libraries available for the host language.

1. Introduction

Contemporary families of secure computation protocols such as secure multi-party computation (MPC) [6] and homomorphic encryption (HE) [14] allow programmers to build applications that operate on data that remains encrypted without decrypting it. While a broad array of general-purpose secure MPC [8, 16, 18] and HE [2, 5, 14, 15] protocols exist (with new ones being introduced regularly), these usually achieve their expressive power at the cost of increased overheads in terms of (1) computation and communication and (2) the mathematical sophistication required to apply them successfully within an application scenario.

Some of the simpler secure computation protocols (such as linear secret sharing schemes [16] or the Paillier cryptosystem [15]) can be used to perform only those computations that can be represented as linear transformations of the encrypted or secret-shared input data. However, even such limited protocols have been deployed in production within suitable scenarios [11]. Furthermore, such protocols *can* be used to implement a broad variety of other functions when working with small finite domains [10] by representing computations as binary matrices that can be applied to one-hot vectors. This technique presents an alternative region within the trade-off space, and exploration of this region when tackling suitable use cases can be facilitated by the right programming tools.

We propose an *embedded domain-specific language* (EDSL) [7] for (1) defining inputs and outputs in a way that enables easy translation to and from their one-hot vector representations and (2) defining functions in a way that enables easy translation to and from binary matrices. This allows programmers to define input and output domains using the host language's type system, and to use all the features, libraries, and paradigms found in the host language when defining functions that operate over these domains. It also allows programmers to compose these capabilities with secure computation protocol libraries available for the host language.

As secure computation protocols mature and find applications, they (1) will need to be incorporated into contemporary software stacks that rely on languages such as Python and JavaScript and (2) will ideally be deployable within cloud-based infrastructure, web browsers, and mobile operating systems. Furthermore, software engineers encountering implemented solutions may be interested in relying on their existing experience and toolchains when auditing them, verifying their correctness, or evaluating their performance. Thus, we illustrate and examine the potential of an EDSL for leveraging one-hot vector and matrix representations by presenting such an EDSL implemented within the Python programming language.

2. Background and Related Work

Numerous frameworks exist for building software solutions that employ secure computation techniques [9, 14], including dedicated programming languages, compilers, and both standalone and embedded DSLs. However, none are specialized exclusively to accommodate an approach that leverages one-hot vectors and binary matrices to implement arbitrary user-defined functions [10]. Techniques based on one-hot vectors are employed within a number of domain-specific [1, 13] and general-purpose [3, 4, 17, 17] secure computation protocols, so there may be value in enabling rapid prototyping of software artifacts that rely on these representations. This work reflects the motivation and structure of prior work on an embedded DSL for building circuits [12] that could be used within garbled circuit protocols [8, 18].

3. Embedded DSL for Leveraging One-Hot Vector and Binary Matrix Representations

The EDSL presented in this work allows programmers to define two categories of constructs. First, it allows programmers to use the host language type system to define and build up *domains* of values that can be automatically converted to and from their one-hot vector representations. A Cartesian product operation over domains is provided, allowing easy composition of domains. Second, it enables the use of any appropriately annotated function as a binary matrix that can be applied to a one-hot vector representation of a value. This can be accomplished either via direct conversion of a function to a corresponding binary matrix or via an operator that performs the matrix operation iteratively (thus avoiding the construction of the entire matrix in memory at any one time).

Adopting the design rationale articulated and demonstrated in prior work on circuit synthesis [12], the EDSL presented in this work is implemented as an open-source Python package.[1] The library provides class definitions for domains, one-hot vectors, and matrices. Its functionalities are achieved by relying on several features of the Python programming language: higher-order functions, inheritance, operator overloading, type annotations, and reflection.

- Each domain consists of one or more component dimensions that must be of a known fixed size and must have a strict order. Domains can be composed via a Cartesian product operator that is implemented by overloading of the built-in `*` operator. The square bracket notation for domain objects is overloaded to allow access to individual values within the domain by their index, while the domain object itself can be applied to a value to obtain that value's one-hot vector representation.
- One-hot vectors are implemented by default as integers (wherein the integer represents the index of the sole nonzero entry in the one-hot vector) to conserve memory. However, also being *iterable*, one-hot vector objects yield the entries of the one-hot vector when iterated. This makes it possible to prepare one-hot vectors for encryption (as presented in Figures 3 and 4). The matrix multiplication operator for one-hot vectors is overloaded to allow composition with encrypted matrix representations (as presented in Figure 5).
- Binary matrix objects wrap their corresponding function objects and can be applied to a one-hot vector instance (or any iterable data structure of the appropriate length and supporting addition and scalar multiplication operations) using the overloaded matrix multiplication operator. This is implemented iteratively in order to conserve memory, but an explicit representation of the matrix can also be obtained. This makes it possible to prepare matrices for encryption (as presented in Figure 5).

4. Example Applications

We illustrate some of the relevant features of the EDSL by considering a simple comparison function. A few examples demonstrate how the EDSL can be used in conjunction with secure computation protocol libraries (either to work with encrypted input and output values or to work with an encrypted function).

4.1 Comparison Operation on a Small Domain of Integers

Figure 1 presents a definition of a comparison function that operates on inputs from the domain of pairs of 8-bit integers and returns outputs within a user-defined enumerated domain consisting of three string values.

```
import matricity

uint8 = matricity.domain(range(2 ** 8))
enum3 = matricity.domain(['less', 'same', 'more'])

def compare(x: uint8, y: uint8) -> enum3:
    if x < y:
        return 'less'
    elif x > y:
        return 'more'
    else:
        return 'same'
```

Figure 1. Simple comparison function for 8-bit integers.

Figure 2 presents how the function can be converted into an abstract binary matrix instance and how this matrix can be applied to a one-hot vector representing a pair of inputs.

```
>>> import matricity
>>> d = uint8 * uint8
>>> v = d((11, 7)) # Input value as one-hot vector.
>>> m = matricity.matrix(compare) # Binary matrix.
>>> w = tuple(m @ v) # Matrix multiplication.
>>> w # Output as one-hot vector.
(0, 1, 0)
>>> enum3[w] # Corresponding output value.
'more'
```

Figure 2. Use of comparison function as a binary matrix.

4.2 Composition with Secure MPC and HE Libraries

Figure 3 presents how the definitions in Figures 1 and 2 can be combined with a library[2] that implements Shamir's secret sharing scheme [16]. Each entry in the input one-hot vector `v` is turned into three secret shares (one for each of three parties). These are grouped into three vectors (each containing the secret shares of the one-hot vector entries intended for one of the three parties). The matrix `m` is applied directly to each of these using the matrix multiplication operator. The entries of the output one-hot vector can then be reconstructed using the interpolation function provided by the library for Shamir's scheme and then converted into a value.

```
>>> import shamirs
>>> (v_a, v_b, v_c) = zip(*[
    shamirs.shares(v_i, 3)
    for v_i in v
]) # Input shares for parties 'a', 'b', and 'c'.
>>> (w_a, w_b, w_c) = (
    m @ v_a,
    m @ v_b,
    m @ v_c
) # Output shares from parties 'a', 'b', and 'c'.
>>> w = tuple(
    shamirs.interpolate(w_p)
    for w_p in zip(w_a, w_b, w_c)
)
>>> w # Reconstructed output one-hot vector.
(0, 0, 1)
>>> enum3[w] # Corresponding output value.
'more'
```

Figure 3. Comparison example within Shamir's scheme.

Similarly, Figure 4 presents how this secure computation can be realized with a library[3] that implements the Paillier cryptosystem [15], applying the matrix `m` to an encrypted input vector `v_e`.

Finally, Figure 5 presents how the example can be modified such that the *function matrix* `m` (rather than the input vector) is encrypted. To make it possible to print a smaller matrix, the `matrix` constructor is given input and output domain parameters that override the annotations of the original function definition in Figure 1.

[1] The library is available at `https://pypi.org/project/matricity/`.

[2] The library is available at `https://pypi.org/project/shamirs/`.

[3] The library is available at `https://pypi.org/project/pailliers/`.

```
>>> import pailliers
>>> sk = pailliers.secret(2048)
>>> pk = pailliers.public(sk)
>>> v_e = [
    pailliers.encrypt(pk, v_i)
    for v_i in v
] # Encrypted input vector.
>>> w_e = m @ v_e # Encrypted output one-hot vector.
>>> w = tuple(
    pailliers.decrypt(sk, w_i)
    for w_i in w_e
)
>>> w # Decrypted output one-hot vector.
(0, 0, 1)
>>> enum3[w] # Corresponding output value.
'more'
```

Figure 4. Comparison example within the Paillier cryptosystem.

```
>>> uint2 = matricity.domain(range(2 ** 2))
>>> d = uint2 * uint2
>>> v = d((3, 2)) # Input value as one-hot vector.
>>> m = matricity.matrix(compare, d, enum3)
>>> for row in m:
...     print(row)
[0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0]
>>> m_e = [
    [pailliers.encrypt(pk, m_ij)) for m_ij in row]
    for m_i in m
] # Encrypted matrix.
>>> w_e = m_e @ v # Encrypted output one-hot vector.
>>> w = tuple(
    pailliers.decrypt(sk, w_i)
    for w_i in w_e
)
>>> w # Decrypted output one-hot vector.
(0, 0, 1)
>>> enum3[w] # Corresponding output value.
'more'
```

Figure 5. Comparison example in which the function is encrypted.

5. Conclusions and Future Work

We have introduced a library that serves as an EDSL for leveraging one-hot vector and binary matrix representations within secure computation protocols. The library allows programmers to leverage the full extent of host language features and to take advantage of polymorphism to compose vector and matrix representations with existing secure computation protocol libraries available for the host language. In the future, the EDSL can be extended to make it easier to predict and profile automatically the storage and computation overheads associated with the one-hot vector representations of values. The library can also be enhanced to support compression techniques for vectors and matrices where they may be appropriate (and compatible with secure computation protocols).

References

[1] A. Abidin, A. Aly, S. Cleemput, and M. A. Mustafa. An MPC-Based Privacy-Preserving Protocol for a Local Electricity Trading Market. In S. Foresti and G. Persiano, editors, *Cryptology and Network Security*, pages 615–625, Cham, 2016. Springer International Publishing.

[2] J. Benaloh. Dense Probabilistic Encryption. In *Selected Areas of Cryptography*, May 1994. URL `https://www.microsoft.com/en-us/research/publication/dense-probabilistic-encryption/`.

[3] E. Boyle, N. Gilboa, and Y. Ishai. Function Secret Sharing. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 337–367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46803-6.

[4] E. Boyle, N. Gilboa, Y. Ishai, and V. I. Kolobov. Information-Theoretic Distributed Point Functions. Cryptology ePrint Archive, Paper 2023/028, 2023. URL `https://eprint.iacr.org/2023/028`. `https://eprint.iacr.org/2023/028`.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Trans. Comput. Theory*, 6(3), July 2014. ISSN 1942-3454. doi: 10.1145/2633600. URL `https://doi.org/10.1145/2633600`.

[6] R. Cramer, I. B. Damgård, and J. B. Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[7] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ, 2011. ISBN 978-0321712943.

[8] O. Goldreich. Cryptography and cryptographic protocols. *Distributed Comput.*, 16(2-3):177–199, 2003. URL `https://doi.org/10.1007/s00446-002-0077-1`.

[9] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237, 2019. URL `https://doi.org/10.1109/SP.2019.00028`.

[10] A. Lapets. Implementing Arbitrary Maps over Small Finite Domains using Ring Addition and Scalar Multiplication. Cryptology ePrint Archive, Paper 2023/1695, 2023. URL `https://eprint.iacr.org/2023/1695`.

[11] A. Lapets, F. Jansen, K. D. Albab, R. Issa, L. Qin, M. Varia, and A. Bestavros. Accessible Privacy-Preserving Web-Based Data Analysis for Assessing and Addressing Economic Inequalities. In *Proceedings of ACM COMPASS 2018: First Conference on Computing and Sustainable Societies*, San Jose, CA, USA, June 2018. URL `https://doi.org/10.1145/3209811.3212701`.

[12] A. Lapets, W. Howe, B. Getchell, and F. Jansen. An Embedded Domain-Specific Language for Logical Circuit Descriptions with Applications to Garbled Circuits. Cryptology ePrint Archive, Report 2020/1604, December 2020. URL `https://eprint.iacr.org/2020/1604`.

[13] J. Launchbury, I. S. Diatchki, T. DuBuisson, and A. Adams-Moran. Efficient Lookup-Table Protocol in Secure Multiparty Computation. *SIGPLAN Not.*, 47(9):189–200, September 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364556. URL `https://doi.org/10.1145/2398856.2364556`.

[14] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj. Survey on Fully Homomorphic Encryption, Theory, and Applications. *Proceedings of the IEEE*, 110(10):1572–1609, 2022. doi: 10.1109/JPROC.2022.3205665.

[15] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[16] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, November 1979. ISSN 0001-0782. doi: 10.1145/359168.359176. URL `https://doi.org/10.1145/359168.359176`.

[17] S. Wagh. Pika: Secure Computation using Function Secret Sharing over Rings. *Proceedings on Privacy Enhancing Technologies*, 2022: 351–377, 10 2022. doi: 10.56553/popets-2022-0113.

[18] A. C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986. URL `https://doi.org/10.1109/SFCS.1986.25`.