Late Breaking Results: COPPER: Computation Obfuscation by Producing Permutations for Encoding Randomly

Kevin Hutto* and Vincent Mooney^*
^School of Computer Science and *School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, Georgia, USA
khutto30@gatech.edu, mooney@ece.gatech.edu

***Abstract*—Deployed embedded devices face security risks due to increased ease of physical access to the devices by unauthorized users. Capable adversaries can intercept a device to recover the data in memory, including results of performed sensitive computations. Device owners require data confidentiality on their physically insecure devices. To satisfy this goal we implement a novel method, COPPER (Computation Obfuscation by Producing Permutations for Encoding Randomly), to create data which never exists on the device digitally in plaintext format and which is subsequently used for computation. In this paper we utilize COPPER to calculate a moving average computation on encoded data.** [1]

***Index Terms*—hardware security, encrypted computations, embedded systems**

I. Introduction and Assumptions

Remotely deployed devices have an inherent lack of security due to their physical vulnerability. Adversaries have the capability to remove a device from service and read the data stored in memory. This data may contain sensitive information which the user does not want leaked, and so the owner may encrypt all data on the device. This is hindering, however, as it may be desired to perform computation on sensed data while on the device, which is hampered by maintaining the data in an encrypted format. Homomorphic encryption schemes have been developed recently which allow computation on encoded data [1]; however, existing homomorphic encryption solutions require intensive time-complexity overhead which to date so far precludes their usage on low-power embedded systems [2].

To perform computations on encoded data in a way distinct from existing homomorphic encryption schemes, we build upon the work we introduced in [3] which performs encoded computations with a fixed, limited number of inputs. This scheme utilizes random permutations to perform calculations on encoded data. We introduce in this paper COPPER (Computation Obfuscation by Producing Permutations for Encoding Randomly), which extends the original scheme to allow more complex computations. The key contribution of this work is the introduction of a novel framework to perform an arbitrary number of computations on encoded data. In this work we perform the calculation of a moving average filter on a stream of data, and we intend to expand the work to implement discrete cosine transforms and more complex finite impulse response filters.

II. Assumptions

We are interested in providing encoded computations which allow a deployed device to compute deterministic time algorithms with an arbitrary depth. We define depth as the number of distinct computations (e.g., adding two numbers takes one computation for a depth of one while adding four numbers takes three computations for a depth of three). To enact the encoded computations we assume there exists a secure server with an insecure communication channel to a deployed device. The deployed device receives sensor data through an integrated analog-to-digital converter (ADC). The sensor is designed such that the data is produced directly in an encoded format which can only be interpreted through usage of a key value. The device will utilize a series of bitstreams sent from the secure server to conduct dynamic reconfigurations of onboard logic in order to perform the computations on encoded data. We assume an adversary can intercept all communications to and from the deployed device and is able to monitor all data stored in memory (RAM) on the device but lacks access to individual registers.

III. Proposed Architecture

A. Encoding Mechanism

To perform encoded computations COPPER requires two broad steps: (1) encoding of input data and (2) computation on the encoded data. To achieve the first step of encoding data we make use of the circuit shown in Fig. 1. This circuit encodes data received through an ADC at the time of data conversion without ever producing unencoded data in memory. The circuit utilizes a 512-bit initial seed, labeled in Fig. 1 as H_0. This seed value is continuously updated via a SHA-3 module configured to run the SHAKE-256 algorithm such that a 512-bit input produces a 512-bit output. Each output of the SHA-3 algorithm, labeled H_j, is used to generate a permutation (LUT_i) of 4-bit values via a hardware rendition of the Knuth Shuffle (Shuffle Unit in Fig. 1). Each permutation LUT_i performs a bijective mapping of the two sets of 4-bit possible unencoded input values $S_i[7:4]$ and $S_i[3:0]$.

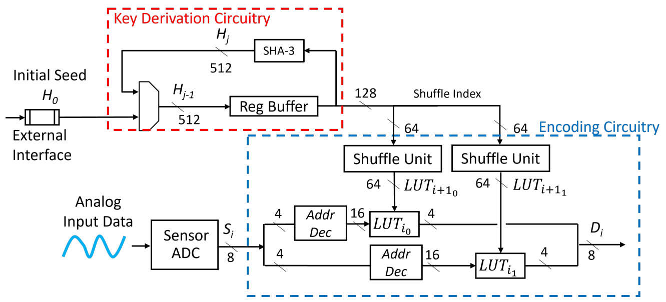

Fig. 1. COPPER Encoding Circuitry

B. Computational Framework

The encoding scheme we introduced in Section III-A allows the server which knows the device's key H_0 to derive all future encoding mappings (LUT_i). As the server knows the potential encodings for each value, the server can create logic tables which will allow the device to perform a look-up using multiple encoding values to obtain an encoded computation result. This is shown in Fig. 2. In Fig. 2 step (a), the server pre-computes the plaintext computation results of a simple addition and replaces the inputs A and B with A' and B', which are derived as the known encodings which will occur from the key H_0. Next, in step (b) a list of unique values is created and permuted. This permuted list of values is used in step (c) as the look up value for the device, the result of the encoding computation,

[1]This work was supported in part by The US Department of Energy Office of Cyber-Security, Energy Security and Emergency Response (CESER), Cybersecurity for Energy Delivery Systems (CEDS) Award to the Georgia Institute of Technology, # DE-CR0000004.

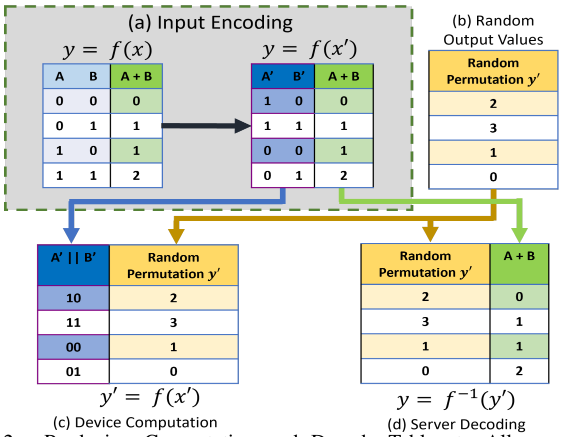

Fig. 2. Producing Computation and Decode Tables to Allow a Single Computation. Encoded Data is Marked as x’.

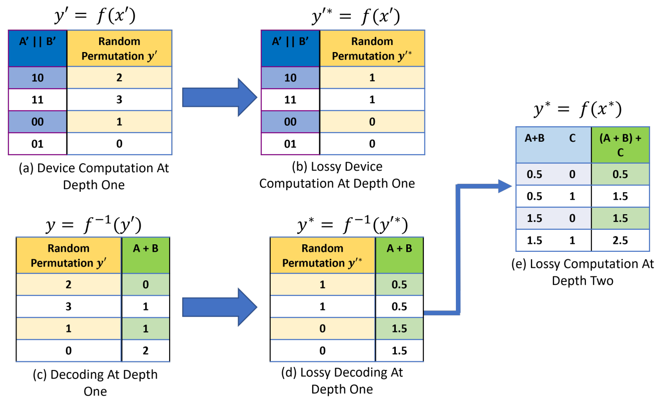
$y' = f(x')$
$y'^* = f(x')$
$y^* = f(x^*)$
A′ || B′ | Random Permutation y'
10 2
11 3
00 1
01 0
A′ || B′ | Random Permutation y'^*
10 1
11 1
00 0
01 0
A+B | C | (A + B) + C
0.5 0 0.5
0.5 1 1.5
1.5 0 1.5
1.5 1 2.5
(a) Device Computation At Depth One
(b) Lossy Device Computation At Depth One
(e) Lossy Computation At Depth Two
$y = f^{-1}(y')$
$y^* = f^{-1}(y'^*)$
Random Permutation y' | A + B
2 0
3 1
1 1
0 2
Random Permutation y'^* | A + B
1 0.5
1 0.5
0 1.5
0 1.5
(c) Decoding At Depth One
(d) Lossy Decoding At Depth One

Fig. 3. Transforming Original Tables to Allow Arbitrary Depth of Computation. Encoded Data is Marked as x’ and Lossy Data as x^*.

and the permuted list is used in step (d) as the input to a table the server will utilize to decode the results of the computation. The Computation Table from step (c) will be transmitted to the device to perform computation, and the Decode Table produced in step (d) will be utilized by the server. The computation shown in Fig. 2 uses a depth of only one operation. To perform multiple operations without loss of data, the naive approach will lengthen the table exponentially with each step as the cumulative bit-width of the input increases with each additional input value. To allow an arbitrary depth of computations, we perform a lossy computation by reducing the bit-length of the output to match the length of the input. The server can then produce Computation Tables which the device will logically chain together, with one table input derived as the output of the previous Computation Table.

An example of how we convert the original tables to lossy Computation and Decode Tables is shown in Fig. 3. Tables (a) and (c) from Fig. 3 are tables (c) and (d) from Fig. 2 respectively. The Computation Table and Decode Table are converted in conjunction. For the Computation Table, we require the output bit-size to be the same size as the input. In the example in Fig. 3, this requires converting the 2-bit outputs of table (a) to 1-bit outputs. This is accomplished by assigning a group of outputs numerically close together (according to the Decode Table output value mapping) the same number. In Fig. 3 the encoded outputs ‘2’ and ‘3’ in table (a), correlating to unencoded values ‘0’ and ‘1’, are both changed to the 1-bit value ‘1’ in table (b). Likewise, the encoded outputs ‘1’ and ‘0’ in table (a), correlating to unencoded values ‘1’ and ‘2’, are reassigned to an encoded value of ‘0’ in table (b). In the Decode Table to accommodate the grouping of these values the unencoded outputs becomes the average of the grouping of each output. This results in decoding to either ‘0.5’ or ‘1.5’ for 1-bit addition. The new output values are then utilized as one input to the next generation of Computation and Decode tables. This process can repeat for arbitrary depth of computation.

The scheme as described so far can lead to large inaccuracies as compared to standard floating point computations. An alternative solution can be accomplished via the assumption that the actual value space needed is much smaller than the allowed value space. For instance, if a particular 16-bit ADC output stream contains sensor values which can always be represented by only 10-bits, then the steps performed in (b) and (d) of Fig. 3 can minimize bit usage by eliminating numbers outside of the expected value space rather than averaging all numbers in a group indiscriminately.

TABLE I
MOVING AVERAGE COMPUTATION ACCURACY

Computation Depth	**Avg. Inaccuracy (%)**	**Max Inaccuracy (%)**
3	20.4	543.7
10	6.2	110.7
20	3.1	87.0
50	1.2	23.3
Overall	7.7	191.21

Inaccuracies in the encoded computation scheme COPPER as compared to floating point computations. Each computation depth option was performed for 10,000 input sequences.

IV. EXPERIMENTATION AND RESULTS

To showcase the proposed encoded computation scheme COPPER, the server and device were implemented in Python 3.10. As validation of the usefulness of the computational framework, moving average filters with varying window sizes were calculated on inputs of random data. The accuracy results of the COPPER computations are shown in Table I. The scheme results in generally low inaccuracies when compared to floating point computations as a golden standard, though very large inaccuracies can occur for individual computations.

V. CONCLUSION AND FUTURE WORK

In this work we have performed computation on an encoded input stream of arbitrary depth in software, calculating a moving average on a stream on data. The computation is conducted such that unencoded data is never present on the device performing computation. Future work will transfer the software implementation into VHDL hardware simulations and explore limiting the inaccuracy of computations as well as explore the applicability of the scheme to more complex computations and various uniform data sets. Furthermore, additional investigation must be conducted to ensure the security validations made in [3] are maintained.

REFERENCES

[1] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford University, 2009, crypto.stanford.edu/craig.

[2] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, “A survey on homomorphic encryption schemes: Theory and implementation,” *ACM Comput. Surv.*, vol. 51, no. 4, Jul 2018. [Online]. Available: https://doi.org/10.1145/3214303

[3] K. Hutto, S. Grijalva, and V. Mooney, “Rancompute: Computational security in embedded devices via random input and output encodings,” in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–8.