A Domain-Specific Language for Constructing and Reasoning About the Security of Garbled Circuits

Matthew Chan Rohit Jha Atyansh Jaiswal

University of California, San Diego

{mattchan,rohitjha,atjasiwa}@ucsd.edu

1. Introduction

Garbled Circuits is a cryptographic protocol which employs idealized boolean circuits that only reveal the result of the computation but nothing about the inputs or the intermediate results. However, despite decades of work by the cryptographic community, garbled circuits remain largely a theoretical construction without a viable, easy-to-use implementation. In this work we present a high level language in the form of an embedded domain specific language in Haskell to both construct and reason about the security of garbled circuits. Our DSL provides an easy way to express circuits and perform secure garbling on them such that they can be used for applications such as secure multiparty computation, or oblivious cloud computation.

2. Attacker Model

We assume two different scenarios where secure computation using garbled circuits would be useful:

1. *Cloud Computation*: Alice wants to perform some computation on some input and would like to use a semi-honest cloud server to perform the computation. Alice would then represent the function in the form of a circuit, garble the circuit and her inputs. The garbled circuit is then evaluated on the cloud server and the encrypted result is returned to Alice, who can then decrypt the result. In this case, the cloud server would be unable to learn anything about Alices secret inputs, or the circuit internals.
2. *Two-Party Computation*: Alice and Bob want to perform some joint computation using their inputs, without revealing anything about their own secret inputs to each other. Alice also possesses the function for the computation which is also secret. Alice would then represent the function in the form of a circuit, garble the circuit and her input. Alice would then send the garbled values to Bob. Bob obtains the garbled value for his input from Alice through oblivious transfer. Bob then evaluates the garbled circuit and shares the result with Alice. Apart from Bob and Alice keeping their values secret from each other, an adversary seeing any communication between the two would not be able to learn anything about their inputs.We assume that Alice and Bob are semi-honest parties.

3. Garbled Circuits

In this section, we provide a high-level description of the process of garbling circuits. We explain how the garbling is performed and how the inputs are applied to the garbled circuit to obtain the final result.

In a two-party garbling process, the inputs from both parties are encrypted by the party that performs the garbling process. For this, the second party has to send its input to the other party and this transfer is called Oblivious Transfer, where the other party is not aware of the other party's actual input. In out implementation, we do not handle Oblivious Transfer and we refer readers to [10] [7] [9] [2] for further details.

Garbling of a circuit involves encrypting each of the circuit's gates' truth tables and permuting the rows of these truth table, thereby ensuring that the truth tables cannot be reverse engineered and an attacker is unaware of the circuit's internals. The garbling process of each gate involves four steps, which are explained in detail in section 5. The garbling of a gate returns a garbled truth table and a lookup table with entries containing rows corresponding to the encrypted output and the actual Boolean output.

4. A Domain-Specific Language for Describing Circuits

To be able to reason about the garbled circuits protocol formally, we develop an embedded domain-specific language (EDSL) of circuits in Haskell and implement garbling as an interpreter for the language. This allows us to apply a range of informal and formal reasoning techniques available to the purely functional programmer to this cryptographic problem.

A design choice is to provide a natural and flexible programming interface for describing and modelling boolean circuits, which suggests a shallow embedding[5].

For example, we would like to be able to express the circuit in Figure 1 as

```
circuit :: Circuit Bool
circuit a1 a2 b1 b2 = do
    gateA_out ← and a1 b2
    gateB_out ← and b1 b2
    gateC_out ← or gateA_out gateB_out
    return gateC_out
```

However, there are a number of limitations to a fully shallow embedding, namely that because programs are represented directly in the metalanguage by their semantics, we cannot subsequently manipulate their ASTs.

Free Monads. We implement our language by the free monads idiom [11], which allows us to obtain a DSL for *circuits* from a suitably parametric datatype of *single logic gates* for free. Intuitively, the type parameter a corresponds to the return type of the monad, and the continuations at each nonterminal node correspond to a monadic action to be executed after the node is evaluated — thus allowing *Free* to traverse the structure by calling the sequence of continuations, and v.v. for construction. A further convenience is that Haskell can derive a *Functor* instance by parametricity[13], leaving us with only a succinct definition to write:

```
data Gate a
  = And Bool Bool (Bool → a)
  | Or Bool Bool (Bool → a)
  | Const Bool a
  deriving Functor
type Circuit = Free Gate
```

While we do not take advantage of their full power, we did find that structuring the language in this way simplifies the construction of evaluation functions (composition of garbling follows from monadic traversals), while enabling the compositional interface we seek.

Free Applicatives. Another interesting structure to explore[1] for the circuits DSL is the *Free Applicative*, which gives an applicative for free from any functor. A consequence of this is that, since applicatives have a weaker interface that disallows dependence on effects, applicative structures support static analysis and inspection[6], which may have different implications for the security of the garbled circuits defined.

5. Garbling Truth Tables

To explain the garbling process and evaluation, we will be using the circuit with a single logical AND gate from Figure 2 as the running example. The truth table for this gate is shown in Table 1.

As mentioned in section 3, the garbling process of a gate involves garbling the gate's truth table in four steps [10]:

[1] Unfortunately instances of the free applicative are not derivable from the free monad directly

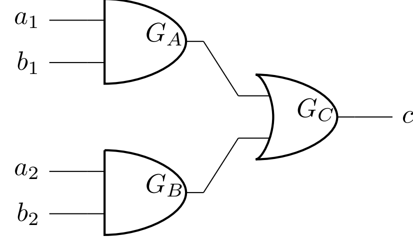

Figure 1: A simple circuit

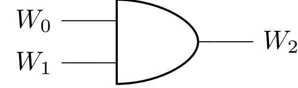

Figure 2: A circuit with only one AND gate

Table 1: Truth table for the AND gate

W_0	W_1	W_2
0	0	0
0	1	0
1	0	0
1	1	1

Step 0: Initialization

Let W_k denote the kth wire of the circuit. For each wire k, we generate a random permutation bit p_k and two 80-bit random keys v_k^0 and v_k^1. Using these, we calculate $w_k^0 = v_k^0||0 \oplus p_k$ and $w_k^1 = v_k^1||1 \oplus p_k$.

Step 1: GTT — *Garbled Truth Table*

To garble the original truth table of a gate, for every wire k we replace all the occurences of 0 with w_k^0 and 1 with w_k^1. This Garbled Truth Table (GTT) is shown in Table 2. In addition, we create a lookup table containing the values $(w_k^0, 0)$ and $(w_k^1, 1)$ as rows, as shown in Table 3 for the AND gate.

Table 2: Garbled Truth Table (GTT) for the AND gate

W_0	W_1	W_2
w_0^0	w_1^0	w_2^0
w_0^0	w_1^1	w_2^0
w_0^1	w_1^0	w_2^0
w_0^1	w_1^1	w_2^1

In our implementation we encapsulate the various stateful operations garbling requires in a monad `Garbled`, defined as:

```
type Garbled a =
```

Table 3: Lookup table for the AND gate

W_2	Output
w_2^0	0
w_2^1	1

$$\begin{aligned}&\mathit{RandT\ StdGen}\\&\quad(\mathit{StateT\ KeySt}\\&\qquad(\mathit{ExceptT\ GarbleError\ Identity}))\ a\end{aligned}$$

The first step of garbling initializes the $KeySt$ map, which maps unique wire tags k to triples of the generated random values (p_k, v_k^0, v_k^1), and returns a *garbled* truth table, encapsulated in the type $GarbledTT$. This is a pair of a truth table with the random value substitutions (of type $Enc = Int$) applied, and the translation tables.

$$gtt :: TT\ Bool \rightarrow Garbled\ (GarbledTT\ Enc)$$

Step 2: EGTT — *Encrypted Garbled Truth Table*

We next encrypt all the entries of the GTT. If a row in the GTT has values (w_i^x, w_j^y, w_k^z), then the corresponding row in EGTT will have the values $(Enc(w_i^x), Enc(w_j^y), Enc(w_k^z))$, where:

$$\begin{aligned}Enc(w_i^x) &= \mathtt{SHA1}(w_i^x||i||x||y)\\Enc(w_j^y) &= \mathtt{SHA1}(w_j^y||j||x||y)\\Enc(w_k^z) &= Enc(w_i^x) \oplus Enc(w_j^y) \oplus w_k^z\end{aligned}$$

Table 4: Encrypted garbled truth table (EGTT) for the AND gate

W_0	W_1	W_2
$Enc(w_0^0)$	$Enc(w_1^0)$	$Enc(w_2^0)$
$Enc(w_0^0)$	$Enc(w_1^1)$	$Enc(w_2^0)$
$Enc(w_0^1)$	$Enc(w_1^0)$	$Enc(w_2^0)$
$Enc(w_0^1)$	$Enc(w_1^1)$	$Enc(w_2^1)$

This amounts to encrypting each row of the table in sequence, which due to our definition of tables requires performing this operation on the transposed table. Intuitively, this is equivalent to[2] the following operation on gtt

$$egtt = transpose \circ mapM\ encryptRow \circ transpose$$

Step 3: PEGTT — *Permuted Encrypted Garbled Truth Table*

According to [10], to further "garble" the truth table, we permute the rows in EGTT based on the permute bits p_i and p_j:

1. If $p_i = 1$: We swap the first two rows with the last two rows
2. If $p_j = 1$: We swap the first and the third rows with the second and fourth respectively

The role of these permutations is to make the position of a certain string in a PEGTT meaningless. However, in our current implementation, we have not followed the above two ways, but instead randomly arranged the rows of EGTT to get PEGTT.

Table 5: One possible permuted encrypted garbled truth table (PEGTT) for the AND gate

W_0	W_1	W_2
$Enc(w_0^0)$	$Enc(w_1^1)$	$Enc(w_2^0)$
$Enc(w_0^0)$	$Enc(w_1^0)$	$Enc(w_2^0)$
$Enc(w_0^1)$	$Enc(w_1^1)$	$Enc(w_2^1)$
$Enc(w_0^1)$	$Enc(w_1^0)$	$Enc(w_2^0)$

This is simply a permutation of $egtt$, expressed as[3]

$$pegtt = pick \circ permutations$$

Finally, the entire garbling operation is expressed as

$$\begin{aligned}&garble :: TT\ Bool \rightarrow Garbled\ (GarbledTT\ HashedBS)\\&garble = gtt \ggg egtt \ggg pegtt\end{aligned}$$

which takes the logical truth table of a given gate and generates a pair of the garbled truth table and the corresponding translation tables.

We note that adding types to the mathematics helped catch many errors, in both our understanding and implementation, as well as the mathematics described in several papers.

6. Evaluating Garbled Circuits

Evaluation of a single garbled gate is simply a sequence of table lookups in the garbled tables, given by a function

$$\begin{aligned}&eval :: GarbledTT\ HashedBS\\&\qquad \rightarrow Bool \rightarrow Bool \rightarrow Maybe\ Bool\end{aligned}$$

To evaluate a garbled gate, the inputs to the gate are first encrypted. The gate performs the computation on these encrypted inputs and generates an encrypted output. This encrypted output is looked-up in the lookup table (that was generated alongside the garbled truth table) and the corresponding Boolean value is returned.

For example, if the two inputs to the AND gate were 0 (False) and 1 (True) on wires W_0 and W_1, then these input values would be encrypted to $Enc(w_0^0)$ and $Enc(w_1^1)$. Next, an entry corresponding to these two values is matched in the

[2] The actual function is more involved, the details of which we omit for the sake of elegance.

[3] Again, eliding details with bounded random number generation, etc

PEGTT and its output, $Enc(w_2^0)$ is returned on wire W_3. This value is an encrypted one and the party interested in determining the actual value decrypts this:

$$\begin{aligned}&Dec(Enc(w_2^0))\\&= Enc(w_0^0) \oplus Enc(w_1^1) \oplus Enc(w_2^0)\\&= Enc(w_0^0) \oplus Enc(w_1^1) \oplus Enc(w_0^0) \oplus Enc(w_1^1) \oplus w_2^0\\&= w_2^0\end{aligned}$$

The party performing the decryption has access to the lookup table and finds out the result corresponding to w_2^0, which in this case is 0 (False).

When a circuit includes multiple gates with outputs of one or more gates flowing into other gates as inputs, then the garbling and evaluation are done for each gate individually. After evaluating a gate, its output is decrypted before passing to the next gate as input. Finally, the Boolean results of the terminal gates in the circuit are returned to the parties involved in the computation as output of the circuit.

6.1 Correctness of Garbling

The correctness of the garbling transformation can be expressed via the commutative diagram shown in Figure 3, following a similar definition of general compiler correctness given by Hutton et al.[8][1].

$$\begin{array}{ccc} Gate & \xrightarrow{garble} & (PEGTT, Lookup) \\ \downarrow{\scriptstyle translate} & & \downarrow{\scriptstyle eval_{garbled}} \\ TT & \xrightarrow{eval} & Result \end{array}$$

Figure 3: Commutative diagram for correctness

In Figure 3, $Gate$ refers to the original logic gate, which compiles to the gate's truth table TT and a pair $(PEGTT, Lookup)$ under interpretations $eval \circ translate$ and $eval_{garbled} \circ garble$ respectively. The functions $eval$ and $eval'$ evaluate TT and $(PEGTT, lookup)$ when provided the input values to produce the same final Boolean result, $Result$.

While this holds from the mathematical definition of garbling, proving this property by program calculation (as demonstrated in [12] and [1]) remains future work.

7. Security of Garbled Circuits

We evaluate the security of our implementation through the security notions defined in [2]. Here, we represent our circuit as f, garbled circuit as F, the inputs as x, encrypted inputs as X, the output as y, the encrypted output as Y and the decryption method d.

7.1 Privacy

A party acquiring (C, X, d) cannot learn anything impermissible beyond that which is revealed by knowing just the final result y. What is permissible to reveal is defined by a side information function $\Phi(f)$. Since our implementation involves using the free monad, we hide away much of the garbling process and the original inputs to the gates. As a result, even if a party accesses this garbled circuit, it is not possible for them to inspect the internals and at every step of the garbling process, we do not preserve the intermediate truth tables.

7.2 Obliviousness

A party acquiring (F, X), but not d, cannot learn (f, x) from the inputs. In our implementation, d is the decryption function combined with the lookup table. Furthermore, since we abstract away the encryption technique and keys for every gate, if the lookup table is not provided either, it is not possible to recover f from F or x from X.

7.3 Authenticity

A party acquiring (F, X) cannot produce a value $Y*$ such that $Y*! = Y$, and $d(Y*)! = \perp$. In our implementation, the only values of Y that can be accepted for a gate are the two values in the lookup table. Since both the keys in this lookup table are 80 bits wide, which are generated everytime a gate is garbled, there are 2^{80} possible keys out of which only two are valid. This makes it difficult to come up with a random $Y*$ that after decryption will exist in the lookup table.

8. Future Work

We intend to create a full secure two party communication protocol, which would involve adding support for oblivious transfer. This would also involve coming up with a useful abstraction to represent communication between two parties, and making sure that no extra information is leaked to the adversary. One direction we were thinking was perhaps to use session types to reason about communication security.

To be able to verify security, we want to take a similar approach as [3] and use refinement types to perform verification of our protocol at the language level.

We can further implement more optimized versions of both garbling and evaluation that have been constructed in the past [7] [9] and verify the correctness and security of those as well.

9. Conclusion

We have created an embedded DSL in Haskell for representing circuits that allows us to garble circuits and evaluate these garbled circuits. We have proved the correctness of our implementation – why evaluation of the garbled circuits is equivalent to evaluation of the original circuit. We have also tested our DSL over circuits involving single Boolean logic gates such as the AND gate, and have found the results to be correct. Furthermore, we have explained why the implementation of garbled circuits in our DSL provides security properties by hiding away the internals of the garbled circuit due to the monadic representation. We believe this

to by a step towards applications such as secure multiparty computation and oblivious cloud computation.

References

[1] P. Bahr and G. Hutton. Calculating Correct Compilers. *Journal of Functional Programming*, 25, Sept. 2015.

[2] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[3] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing tls with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459. IEEE, 2013.

[4] P. Capriotti and A. Kaposi. Free applicative functors. *arXiv preprint arXiv:1403.0749*, 2014.

[5] J. Gibbons and N. Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *ACM SIGPLAN Notices*, volume 49, pages 339–347. ACM, 2014.

[6] M. Hauck, S. Savvides, P. Eugster, M. Mezini, and G. Salvaneschi. Securescala: Scala embedding of secure computations. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 75–84, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4648-1. doi: 10.1145/2998392.2998403. URL `http://doi.acm.org/10.1145/2998392.2998403`.

[7] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[8] G. Hutton and J. Wright. Compiling Exceptions Correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, Stirling, Scotland, July 2004. Springer.

[9] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 285–300, 2012.

[10] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA, 2004.

[11] W. Swierstra. Data types à la carte. *Journal of functional programming*, 18(04):423–436, 2008.

[12] W. Swierstra and T. Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36. ACM, 2007.

[13] P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.