Poster: Framework for Semi-Private Function Evaluation with Application to Secure Insurance Rate Calculation

Daniel Günther
guenther@rangar.de
TU Darmstadt
Darmstadt, Germany

Ágnes Kiss
kiss@encrypto.cs.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Lukas Scheidel
lukas.scheidel@web.de
TU Darmstadt
Darmstadt, Germany

Thomas Schneider
schneider@encrypto.cs.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

ABSTRACT

Private Function Evaluation (PFE) allows two parties to jointly compute a private function provided by one party on the secret input of the other party. However, in many applications it is not required to hide the whole function, which is called Semi-Private Function Evaluation (SPFE). In this work, we develop a framework for SPFE which allows to split a function into public and private parts. We show the practicability of using SPFE in a real world scenario by developing a car insurance application for computing user-specific tariffs. We evaluate the performance of our SPFE framework on this concrete example which results in a circuit consisting of 377 032 AND gates which improves over PFE by a factor of 9x.

CCS CONCEPTS

• **Security and privacy** → **Privacy-preserving protocols**; **Privacy protections**.

KEYWORDS

Private function evaluation, universal circuit, secure function evaluation

ACM Reference Format:
Daniel Günther, Ágnes Kiss, Lukas Scheidel, and Thomas Schneider. 2019. Poster: Framework for Semi-Private Function Evaluation with Application to Secure Insurance Rate Calculation. In *2019 ACM SIGSAC Conference on Computer& Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3319535.3363251

1 INTRODUCTION

The age of digitalization offers many benefits in areas like communication, automation technology and the entirety of science. While this, on the one hand, provides a lot of comfort, it can be a massive threat to the privacy of users. If we take a look at car industry where digital solutions like autonomous cars are being developed and may be ready for mankind in a few decades, we can easily see the practicability of digital modernization. These cars require a massive number of sensors to collect data which is then used by the car in order to operate properly. Likewise, the data measured by the sensors may be interesting for car insurance companies as well. For instance, car insurance companies may be able to perform a much more fine-grained insurance rate calculation using the data provided by the sensors. However, most of the data (e.g. previous accidents) may be privacy-sensitive to the users of the car. Additionally, car insurance companies have a massive interest in keeping their tariff calculation details private since they are the base of their business.

Private Function Evaluation. In order to protect sensitive data and still being able to compute a private functionality, *Private Function Evaluation* (PFE) can be used, that is a generalization of *Secure Function Evaluation* (SFE). SFE allows two parties to jointly compute a public function f on their private inputs and obtain no information but the result f. In PFE the function f is also private information and is provided by one of the parties. Private function evaluation can be realized using SFE of so-called *Universal Circuits* (UCs) [12]. UCs are special boolean circuits that can be programmed to compute any boolean function $f(x)$ up to size n by defining programming bits p, i.e. $UC(p, x) = f(x)$. The function holder inputs the programming bits p to the SFE protocol while the other party inputs its private data x as before. The function is completely hidden (beyond its size n) from the other party since it completely relies on the private programming bits p of the function holder. Note, a function may result in a huge UC which is impractical with regards to runtime complexity as a UC for a function of size n has asymptotic size $\Theta(n \log n)$ [13]. However, in many applications it is not required to hide all parts of the function. For instance, a car insurance company is not required to hide that their tariffs will be more expensive when the driver's age is below 25 since this information is publicly known. Paus et al. [11] call this kind of functions *Semi-Private* and the evaluation of semi-private functions is called *Semi-Private Function Evaluation* (SPFE). They implemented a compiler for SPFE as an extension of the Fairplay SFE framework [10], where the semi-private functions are composed of so-called *Privately Programmable Blocks* (PPB). A PPB can be programmed to implement one function out of a set of functions. Nevertheless, the Fairplay SFE framework is no longer supported and thus, the compiler for SPFE might be

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3363251

considered obsolete. To the best of our knowledge, this is the only implementation of Semi-Private Function Evaluation to date.

Our Contributions. In this work, we develop a demonstrator that shows the practicability of SPFE using SFE of either plain circuits or UCs for public or private parts of the function, respectively. We develop a SPFE framework that takes as input a set of public and private sub-functions written in ANSI C, compiles these into circuits, and combines these functions with our merger into one semi-private function. We evaluate this function using the ABY-SFE-framework of [5]. As example application, we use our framework for a scenario where a car insurance company wants to compute the car insurance rate for users. We measure the runtime and communication complextiy for our car insurance company application to conclude that SPFE is efficient enough for real-world deployment.

2 SEMI-PRIVATE FUNCTION EVALUATION FRAMEWORK

Our SPFE framework securely computes a semi-private function f in a two-party computation setting using Yao's Garbled Circuits [14] or the GMW protocol [6]. Therefore, it takes f as input separated in multiple files each containing one sub-function declared as either private (f_{priv}) or public (f_{pub}). Additionally, it takes a so-called *Merge file* describing how the sub-functions are composed in order to correctly compute f. We transform the sub-functions f_{priv} and f_{pub} into boolean circuits C_{priv} and C_{pub} using the CBMC-GC framework [4, 8]. These boolean circuits are given in the Bristol circuit format [2]. Subsequently, we use the UC framework of [3, 7, 9][1] to process the private boolean circuits C_{priv} into UCs denoted as UC. The UC framework outputs for each circuit in C_{priv} the UC and the corresponding programming bits p.

In the next step, we build one circuit C from all circuits in C_{pub} and UC. Thus, we develop a merger that takes C_{pub}, UC, p, and the Merge file as input. In the Merge file, we describe the topology of the resulting circuit C, namely the input and output wires, as well as how the sub-circuits are connected. Consequently, C consists of both boolean circuits and UCs and hence, the programming bits p need to be adapted depending on the circuit description in the Merge file[2].

Lastly, we use the ABY-SFE-framework [5] to securely evaluate the public circuit C given the private inputs x and (y, p) of the client and the server, respectively. Note, it is unusual that the server does hold any input in PFE. However, the circuit C contains public parts as well in which the server cannot hard-wire any inputs securely. As a consequence, we allow the server to hold private input data y, too. An abstract overview about the internal process of our SPFE framework is given in Figure 1.

3 IMPLEMENTATION

As described in §2, our framework requires a Merger to compose the public boolean circuits and UCs into one circuit. The merger takes a set of public and private circuits as well as the mentioned *Merge File* that descibes the circuit's topology as inputs. After the merging process, we get a circuit in the ABY circuit format [5] that contains all private and public sub-circuits in the topology specified within the Merge File. We implement the merger in C++ and validate it with test cases using the *GoogleTest framework* [1].

Our Merger parses the input Merge File and creates two files, one for the circuit and one for the programming bits. Thereby, the Merger copies all the sub-circuits into the resulting circuit file in the given topology. Consequently, we adapt the wire numbers within the sub-circuit files accordingly.

Automatization. We automate the processing of the sub-functions using a Shell script. Holding a project folder consisting of the Merge file as well as all the public and private sub-functions implemented in C, our Shell script performs the following steps:

(1) Apply CBMC-GC [4] to each of the sub-functions
(2) Run the UC framework [3, 7, 9] to translate all private sub-circuits into UCs
(3) Combine all the UCs and public sub-circuits to one circuit with our Merger
(4) Run the ABY framework [5] to on the combined circuit via SFE.

Usually, the circuit transformation is processed once while the SFE application (step 4) is executed multiple times. Therefore, our Shell script provides options to run each transformation step individually or any combination of them.

4 EVALUATION

To evaluate our framework we choose a real world application to show feasibility of runtimes and communication. Car insurance companies calculate their tariffs based on diverse comprehensive information of their clients which may be extremely sensitive. Thus, the data needed by car insurance companies are valuable and often reluctantly disclosed by the clients. However, for car insurance companies it would be even better to obtain more data from their clients as these enable them to calculate insurance rates in a more fine-grained way. Also data collected by smart cars, which measure the quality of a client's driving style, can be used to provide more user-specific tariffs. Therefore, we chose to design a car insurance company application to benchmark our SPFE framework.

To provide a reasonable real world scenario, we looked at several German car insurance companies to obtain an insight at how they calculate their tariffs. On this base, we design our own tariff calculation function that results in tariffs within the span of the researched car insurance companies. We split our function into 15 sub-circuits from which 6 are private in the sense that they hide the internal computation (aside from the client's input that is protected by SFE anyway).

Size of our Circuit. We apply CBMC-GC [4] and the UC compiler [3, 7, 9] for the private circuits to the 15 sub-functions and combine the resulting sub-circuits to one circuit with our merger. This results in a circuit with 377 032 AND gates and 1 087 260 XOR gates. For comparison, a whole UC on the same function would result in 3 389 525 AND gates and 9 936 186 XOR gates, i.e. we improved the circuit size by factor 9x.

Test Environment. We run the benchmarks on two Arch Linux system with an Intel Core i9-7960X @2.8 GHz processor and 128 GB RAM each. The two systems are connected via a network that

[1] https://encrypto.de/code/UC
[2] An example merge file is given and documented in our Git respository https://github.com/danguenther/spfe-framework.

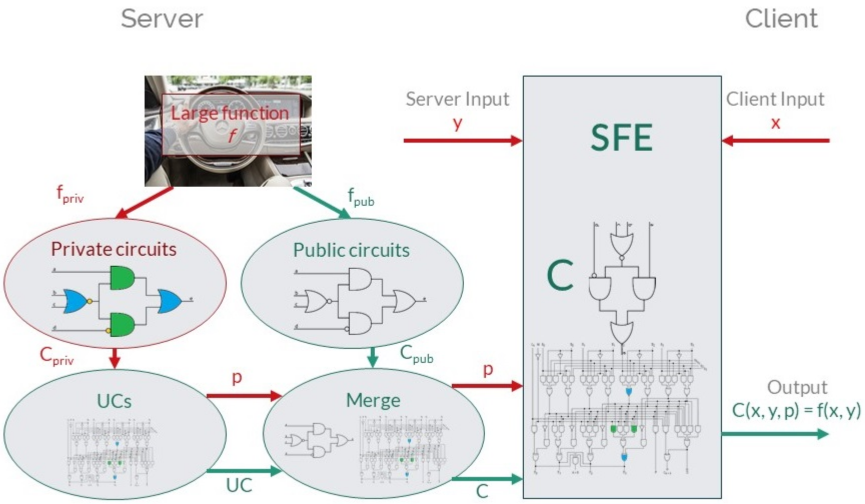

Figure 1: Overview of our SPFE framework. Public information is in green and all protected information is in red.

is configured as a WAN setting (100Mbit/s Bandwidth, 100ms RTT) which resembles a realistic scenario for a car insurance company deploying SPFE. All runtime tests are run from the console without running any other programs in parallel.

ABY Runtime and Communication. When using the constant round Yao's protocol [14], the setup phase takes 1.8s, whereas the online phase takes 2.6s and 17.5 MB communication. When using the multi-round GMW protocol [6], the setup phase takes 1.8s, whereas the online phase takes 30 minutes and 22.1 MB due to the high round complexity of 34 543 rounds. We see that our SPFE framework is practical for real-world applications when using Yao's garbled circuits [14].

Acknowledgements. This project was supported by the DFG as part of project E4 within the CRC 1119 CROSSING and project A.1 within the RTG 2050 "Privacy and Trust for Mobile Users", and by the BMBF and HMWK within CRISP.

REFERENCES

[1] 2008. Googletest - Google Testing and Mocking Framework. https://github.com/google/googletest. Accessed: 2019-03-01.
[2] Victor Arribas Abril, Pieter Maene, Nele Mertens, and Nigel Smart. [n.d.]. Bristol Fashion MPC Circuits. https://homes.esat.kuleuven.be/~nsmart/MPC/. Accessed: 2019-08-23.
[3] Masaud Y. Alhassan, Daniel Günther, Ágnes Kiss, and Thomas Schneider. 2019. Efficient and Scalable Universal Circuits. Cryptology ePrint Archive, Report 2019/348. https://ia.cr/2019/348.
[4] Niklas Büscher and Stefan Katzenbeisser. 2017. *Compilation for Secure Multi-party Computation*. Springer.
[5] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*. The Internet Society.
[6] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play any Mental Game or a Completeness Theorem for Protocols with Honest Majority. In *STOC'87*. ACM, 218–229.
[7] Daniel Günther, Á. Kiss, and T. Schneider. 2017. More Efficient Universal Circuit Constructions. In *ASIACRYPT'17*. Springer, 443–470.
[8] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. 2012. Secure Two-party Computations in ANSI C. In *CCS'12*. ACM, 772–783.
[9] Á. Kiss and T. Schneider. 2016. Valiant's Universal Circuit is Practical. In *EUROCRYPT'16*. 699–728.
[10] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay—a Secure Two-party Computation System. In *USENIX'04*. USENIX Association, 20–20.
[11] Annika Paus, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Practical Secure Evaluation of Semi-Private Functions. In *ACNS'09 (LNCS)*. Springer.
[12] L. G. Valiant. 1976. Universal Circuits (Preliminary Report). In *STOC'76*. ACM, 196–203.
[13] Ingo Wegener. 1987. *The complexity of Boolean functions*. Wiley-Teubner.
[14] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS'86*. IEEE, 162–167.