Update: Lessons Learned from Implementing a Privacy-Preserving Smart Contract in Ethereum

Andreas Unterweger, Fabian Knirsch, Christoph Leixnering,
Dominik Engel

November 27, 2017

In October 2017, we submitted a paper titled *Lessons Learned from Implementing a Privacy-Preserving Smart Contract in Ethereum* [1] to the *Blockchains and Smart Contracts workshop (BSC)*[1] for review. Due to the fast-paced changes in the Ethereum network, especially the ones introduced through the so-called *Byzantium Hard Fork*[2] from October 16, 2017, our results as well as two of our conclusions require an update.

Since we were unable to re-evaluate all of our results and rewrite the paper in time before the submission deadline of the workshop, this technical report is intended to provide (i) a summary of all changes that we made to the original paper [1] and (ii) the new version of our paper with the updated results and conclusions.

The updated version of our paper differs from the original as follows:

- Due to the **change in fees**, we re-deployed our smart contract and updated the table of costs as well as all other cost-related calculations, including the total costs. Since the latter are now about only half of what they used to be in our original test, we also updated the cost-related conclusions. More precisely, we remarked that the costs are now only about one order of magnitude off instead of two. In addition, we added a new bullet point to the list of traps and pitfalls mentioning the change in fees with a refernce to the date and costs of our previous test. Note that between our re-evaluation on November 12 and the time of writing – a period of two weeks – the price of Ether increased further by more than 50%.

- As the EVM gained **new low-cost instructions for cryptographic primitives**, in particular operations relevant for RSA and ECC, we weakened our conclusion to state that not all, but only certain types of privacy-preserving protocols are costly to implement in Ethereum. More precisely, for homomorphic encryption, embeddings and other cryptographic methods which allow computations like the Euclidean distance in the cipherspace domain, our original conclusions still hold.

[1] `http://www.ntms-conf.org/ntms2018/call-for-workshops/bsc2018`

[2] `https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/`

- Due to the significantly **shorter transaction delays**, we updated the correspondingly named bullet point in our list of traps and pitfalls. Besides changing the raw time values to what we observed, we removed our comment on high gas prices relative to the transaction time. It is no longer valid considering the improved transaction delays.

- Since we no longer observed differences between the costs in the test network and the productive Ethereum blockchain, we removed the bullet point on **staging costs** in our list of traps and pitfalls.

In addition, we deployed a slightly modified version of our smart contract (compared to our original paper) which allows it to be reused directly by others for the purpose of testing, saving the repeated deployment costs. While the code changes are negligible, also in terms of total cost, there is now a distinction between the deployment and the execution costs of the smart contract. This is reflected in the paper, together with the updated block number and address of our deployed smart contract.

The updated version of our paper can be found on the next pages. This technical report will be updated once the original paper is either accepted to or rejected from the workshop. Please direct any questions to andreas.unterweger@en-trust.at. A document highlighting the changes between the original version of the paper and the new one is available upon request.

References

[1] A. Unterweger, F. Knirsch, C. Leixnering, and D. Engel, "Lessons Learned from Implementing a Privacy-Preserving Smart Contract in Ethereum," in *9th IFIP International Conference on New Technologies, Mobility and Security (IFIP NTMS)*, submitted.

Updated: Lessons Learned from Implementing a Privacy-Preserving Smart Contract in Ethereum

Andreas Unterweger, Fabian Knirsch, Christoph Leixnering, Dominik Engel
Center for Secure Energy Informatics, Salzburg University of Applied Sciences, Urstein Süd 1, Puch/Salzburg, Austria
Email: andreas.unterweger@en-trust.at

Abstract—**Real-world smart contracts which preserve the privacy of both, their users and their data, have barely been proposed theoretically, let alone been implemented practically. In this paper, we are the first to implement a privacy-preserving protocol from the energy domain as a smart contract in Ethereum. We elaborate on and present our implementation as well as our practical findings, including more or less subtle traps and pitfalls. Despite major optimizations to our implementation, we find that while it is currently possible, it is not feasible to implement certain types of privacy-preserving protocols in the Ethereum blockchain due to the high cost of operation and the lack of privacy by design.**

I. Introduction

Since the inception of Bitcoin [1], [2], the capabilities of blockchains have been significantly expanded [3]. Three noteworthy examples are Zerocash [4], Ethereum [5] and the protocol proposed by [6]. While Zerocash and the protocol from [6] extend the privacy properties of blockchains limited in their computational capabilities, Ethereum allows for much more complex (Turing-complete) operations, referred to as *smart contracts* [7], [8]. However, Ethereum by itself is not privacy-preserving since all calculations are publicly visible to all participants, similar to Bitcoin [9].

In [10], an approach for privacy-preserving smart contracts is presented, but requires a trusted party, which in many use cases is undesired. As an alternate approach to achieve privacy for complex operations in existing blockchains, an additional layer of cryptography can be added [11]. Even though some authors propose privacy-preserving smart contracts, e.g., [11]–[13], none of them show practical implementations. While [12] does not provide an implementation, [11] only discusses a game of rock-paper-scissors as a toy example and [13] does not consider privacy.

In this paper, we implement a smart contract from the energy domain initially proposed in [14], which is both, privacy-preserving and of a practical level of complexity. This protocol solves a problem in a way that is representative for state-of-the-art privacy enhancing technologies, as will be shown. To the best of our knowledge, ours is the first paper to describe such an implementation. In order to facilitate similar implementations for others, we provide insights from our hands-on experiences in developing and deploying a smart contract in the (public) Ethereum blockchain. This includes a discussion ranging from implementation pitfalls to deployment and execution costs.

This paper is structured as follows: In Section II, we describe the privacy-preserving protocol that we implement in Ethereum in Section III. In Section IV, we summarize our observations from the implementation and evaluation processes, before concluding the paper in Section V.

II. Privacy-preserving load profile matching

In [14], a protocol for privacy-preserving smart grid tariff-decisions is described. The protocol allows a customer to choose an optimal tariff based on their energy consumption from a variety of tariffs offered by different utility providers. Among other guarantees, the implementation of the protocol assures that neither the customer's energy consumption data nor their final tariff choice is revealed.

This is achieved by the use of embeddings [15], oblivious transfer [16], and commitment schemes [11]. Embeddings allow transforming energy consumption data into a binary representation that is hard to reverse, but still allows for the required comparisons. For permanent storage, transparency and immutability this comparison is handled in a smart contract which returns (simplified) a once usable pointer to the best-matching tariff to the customer. Oblivious transfer is then used (off-chain) by the customer for retrieving the actual tariff without revealing the decision to the utility provider. To guarantee non-repudiation, i.e., values that are not fully revealed at the time of submission cannot be changed later, a commitment scheme based on cryptographic hashes is used. A formal description of the protocol can be found in Appendix A. Further details of the protocol as well as a detailed privacy analysis can be found in [14], [17].

The smart contract implementing the protocol must be structured as follows [14]:

- `create(sender, energy_data_hash)`: This method is called upon the initial creation of the smart contract. The customer (sender) calculates an embedding of the energy consumption data used as the basis for matching and provides the hash to the smart contract as a commitment.
- `commit(sender, tariff_data_hash)`: This method is called after `create` by each utility (sender) wanting to offer a tariff. The utility calculates an embedding of the energy consumption data corresponding to each of its offered tariffs and submits the hashes as a commitment. This method can only be called once per utility for this instance of the smart contract.

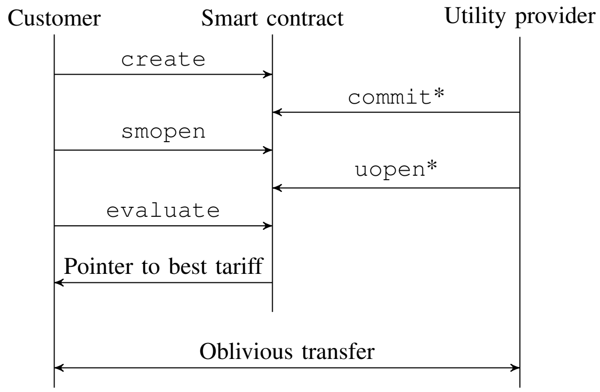

Fig. 1. Sequence diagram that illustrates the function calls between the participants of the implemented protocol. * denotes that every utility provider calls the function.

- `smopen(sender, energy_data, random_number)`: This method is called by the customer (sender) in order to open the commitment with the required random number. It is verified whether the commitment is valid and, if so, the embedded energy data is stored for later matching.
- `uopen(sender, tariff_data, random_numbers)`: Similar to `smopen`, each utility (sender) opens its commitment.
- `evaluate(sender)`: This is method is called by the customer (sender) after `smopen` and `uopen` have been called by all participants. The best-matching tariff is determined using the embedded values and a once-usable pointer for the oblivious transfer is returned.

The sequence of calls to functions of the smart contract is illustrated in Figure 1. While the smart contract in [14] is only provided in pseudo-code, here we describe an actual implementation in Ethereum in the following section.

III. Implementation in Ethereum

We implemented the smart contract from Section II in Solidity 0.4.0 (https://solidity.readthedocs.io/en/develop/), which allows to design such contracts with public and private methods and provides a set of basic data types. Programs compile to EVM (Ethereum Virtual Machine) code that can be deployed as a smart contract into the Ethereum blockchain [3].

Each smart contract and each caller of such a contract is assigned a unique address. Smart contracts can communicate with each other through messages. The caller of a function, i.e., the sender of a message, is implicitly available in the function body as `msg.sender` in Solidity and therefore does not need to be passed as an additional argument. Thus, the signatures of the implemented smart contract differ from the ones from Section II in that they do not need an explicit `sender` parameter.

In order to call smart contracts, each party of the protocol needs to create an Ethereum account. For the proposed protocol, this means that the customer as well as each individual utility provider are assigned a unique ID. For privacy, the customer can either create one account for each execution of the smart contract or change its ID frequently so that it cannot be tracked. Conversely, the utility providers should (but not necessarily need to) re-use their IDs for transparency and accountability for their offered tariffs.

The data types used in the implemented smart contract are

- `uint`: a 256-bit unsigned integer type used to store distance values. When smaller values are stored, types such as `uint8` or `uint16` are used;
- `bytes32`: an array of 32 bytes, i.e., 256 bits, size used to store the output of a cryptographic hash function;
- `address`: a special data type used to store the address of a message caller;
- `mapping`: a hashmap used to store the commitments of each utility provider's tariff-related data;

and arrays of the aforementioned types, e.g., `bytes32[]`.

The actual implementation closely follows the pseudo-code from [14]. Since storing and executing EVM code incurs fees depending on the data size and type (specified in detail in [5]), some implementation details differ from the original version in order to reduce these costs. The three major optimizations are:

- **Use of built-in cryptographic primitives**: Since the EVM provides a relatively cheap instruction for computing a SHA-3 hash, the latter is used in favor of SHA-2. The availability of cryptographic primitives for smart contracts is favorable for the implementation of privacy-preserving protocols.
- **Mathematical tweaks**: Calculating the absolute difference between two binary vectors is equivalent to an element-wise `xor` operation. The Hamming distance of the difference vector can be calculated by counting the number of bits set to one in this vector, which can be simplified with a lookup table for groups of elements. The lookup table size is a trade-off between storage costs (larger tables require more storage) and execution costs (larger tables require less lookups and iterations). Using such mathematical tweaks can be essential to keep costs low, similar to any other programming language.
- **Avoiding unnecessary storage**: `uopen` computes and saves distances to the customer's embedding instead of saving the larger embedding data in a variable and computing the distance at the call of `evaluate`. Conversely, `evaluate` just computes the smallest distance from the ones stored. This reduces costs since the large embeddings (kilobytes in size each) from the utility providers do not need to be saved in the blockchain. In general, refining the program flow in order to avoid storage costs is desirable.

Note that, in addition to these optimizations as well as further minor optimizations, embeddings are already calculated off-chain as originally proposed in [14] and therefore induce no execution costs. In order to illustrate the syntax as well as the aforementioned optimizations, Appendix B shows sample code in Solidity for selected portions of the implemented smart contract. The full source code is available at https://www.en-trust.at/downloads/.

To represent embeddings of 8192 bits size in Solidity, there are several possibilities, e.g., using fixed-sized arrays of base data types like `uint256` or `bytes32`, both of which are capable of storing 256 bits. We found that operations on these types are significantly cheaper than on variable-length types like `string` or `bytes`, which is why we used fixed-sized arrays of `bytes32` for our main computations.

As mentioned above, each operation in the EVM induces costs in a unit called *gas*. These costs are a fee paid to the miner for executing the code of the smart contract. The fee is payed in the built-in crypto-currency Ether. The conversion rate between gas and Ether is set by each miner individually before executing the smart contract. For testing smart contracts, a test net exists which allows executions free of charge. In contrast, using the Ethereum blockchain induces non-negligible costs. Thus, in the following section, we implemented, deployed and assessed our smart contract in the (pay-to-use) Ethereum network.

IV. Lessons learned

This section describes the lessons learned from deploying the smart contract from Section III to the Ethereum network. We determine the total amount of fees required to deploy and run the smart contract with example data from [17]. Furthermore, we report our insights from this experiment with respect to the practicability of running a privacy-preserving smart contract in Ethereum.

We used the default tariffs from [17] to get representative example data for calling the smart contract. For simplicity, each of the three utility providers (denoted U1 to U3) is assigned two of the six sample tariffs each (denoted T1 and T2). The smart contract was deployed into the Ethereum blockchain in block 4538517 with address `0x8e9c8624cb7986f7adc170bd7d1374dfd9f600bd` and its reusable part in block 4438303 with address `0xfad26e55c6a616c57e428e1beb6cdc6b8c42bb10`. Note that this smart contract is not limited to specific customers, utilities or tariffs and is available to anyone for running the protocol. The source code and tariff data are available at https://www.en-trust.at/downloads/.

Table I summarizes the processing costs for our experimental setup in gas as well as the respective value in ETH and € for convenience. We used http://etherscan.io to determine the actual gas costs for each call to the smart contract. The cost for 1 unit of gas was on average 20 Gwei = $20 \cdot 10^{-9}$ ETH, which is also the average gas cost listed by https://ethstats.net/ and others at the time of execution for "fast" processing, i.e., the rate at which the transaction only takes little time to mine. The exchange rate between € and ETH as of November 12, 2017 is 263.60 €/ETH according to https://www.coingecko.com/en/price_charts/ethereum/eur. Note that we execute the smart contract only once. While future fees (and exchange rates) may differ, their order of magnitude can be determined by our experiment.

As can be seen from Table I, the deployment of the smart contract through `create` by the customer requires 4.75 € worth of gas. `commit` is called by each utility once, as described in Section II, and costs each utility 0.51 € on average for two tariffs. `smopen` is called by the customer at the price of 3.90 €. Each utility needs to call `uopen` for each tariff individually, resulting in costs of 6.47 € on average per utility. Finally, `evaluate` is called by the customer at the rate of 0.19 €. In total, the customer needs to pay 8.83 € and each of the three utilities pays on average 6.98 €. This results in total costs of 29.77 € for deploying and executing the smart contract. While it is clear that this is much more than any party in this protocol would be willing to pay (to potentially find a better energy tariff likely to reduce the energy bill by only a fraction of this amount), these costs may be acceptable in other use cases.

In addition to these execution costs, the deployment of the reusable smart contract costs 1061688 gas (ETH 0.010616880 at 10 Gwei), which is equal to 2.80 €. Note that these costs occur only once, meaning that future use of this smart contract only requires the execution costs for one run of the protocol.

Even though it may be possible to further reduce the amount of required gas in our implementation, the actual costs in € above are at least one order of magnitude higher than would be acceptable for this use case. In addition, further optimizations would be time-consuming and therefore also cost-intensive, illustrating that the total costs for implementing and executing privacy-preserving protocols are far too high for practical use.

Despite the popularity of Ethereum, we would like to point out that this issue is not specific to Ethereum. There are privacy-preserving blockchains, e.g., HAWK [10], but no public implementation of them is available at the time of writing (November 2017). We expect the costs to decrease significantly when using blockchains which provide privacy by design, since many computations are performed implicitly.

It is crucial to note that our conclusions regarding costs are also not specific to the protocol from [14] that we implemented. The embeddings used therein are, in terms of ciphertext size and complexity, comparable to state-of-the-art cryptography in the energy domain which allow ciphertext operations [18], [19]. For example, comparisons to the more commonly used Paillier cryptosystem [20] as in [14], [17] show that embeddings require fewer computations and less overhead, i.e., the costs for using the Paillier cryptosystem or similar privacy-preserving technologies are expected to be even more costly than the embeddings used in our evaluation.

Although it may be possible to redesign the protocol, e.g., such that `uopen` can be performed once for multiple customers, this would impact privacy and a rigorous security re-evaluation of the protocol would be required. We do not consider it to be feasible to modify proven and well-tested protocols so that they minimize (gas) costs. Thus, we conclude that it is currently not feasible to use Ethereum to implement state-of-the-art privacy-preserving protocols. This may change in the future, if more cheap EVM instructions for cryptographic primitives, like the one for built-in SHA-

TABLE I

COSTS FOR GAS FOR EACH FUNCTION OF THE IMPLEMENTED SMART CONTRACT FOR THE DEFAULT TARIFFS USED FOR EVALUATION [17]. THE GAS PRICE IN ETH AS WELL AS THE CONVERSION RATE FROM ETH TO € IS FROM THE DATE OF EXECUTION, SUNDAY, NOVEMBER 12, 2017.

Method	**Gas U1T1**	**Gas U1T2**	**Gas U2T1**	**Gas U2T2**	**Gas U3T1**	**Gas U3T2**	**Price (ETH)**	**Price (€)**
`create`	900448						0.018 008 960	4.75
`commit`	97241		97241		97241		0.005 834 460	1.54
`smopen`	739364						0.014 787 280	3.90
`uopen`	618517	618236	603261	617980	603517	619324	0.073 616 700	19.41
`evaluate`	35167						0.000 703 340	0.19
Sum							0.112 950 740	29.77

3 computation, are offered. Until then, the costs exceed any practical bounds for many potential use cases.

Apart from costs, we observed the following traps and pitfalls that we would like to share with future developers of privacy-preserving smart contracts:

- **Determining gas limit:** Each call to a smart contract is assigned a caller-specified gas limit, which, once reached, terminates the execution of the EVM. Since the amount of gas the caller is willing to spend determines whether and when a transaction is executed by a miner, in practice, it is hard to set proper limits. One needs to find a tradeoff between cost-effectiveness and throughput. Since gas is a non-refundable fee to the miner, setting the gas limit too low results in a costly and aborted function call.
- **Undocumented language limitations:** At the time of writing (November 2017) Solidity comes with a number of limitations, only some of which are documented. Examples are very limited stack size, which is problematic when dealing with multiple ciphertext variables, e.g., embeddings.
- **Handling large data:** Ethereum is not inherently designed to handle larger amounts of data within smart contracts. As discussed above, custom solutions for this are necessary, which might be cost intensive.
- **Transaction delay**: It takes between around 15 and 90 seconds for each transaction to be confirmed. Reducing the gas price and limit would increase this delay further, which would be unacceptable for this use case. Conversely, increasing the gas price and limit would make the transactions even more expensive without significantly reducing the delay.
- **Changing fees:** We deployed our smart contract multiple times during October and November 2017. Costs changed significantly each time (e.g., on October 15, we payed approx. 60 €). This makes it hard to predict the actual costs for smart contract users over longer periods of time.

In summary, smart contracts are a promising way to implement complex privacy-preserving protocols without the need to rely on a single trusted third party. However, for practical use, there are still many limitations to overcome, especially when advanced state-of-the-art privacy-enhancing technologies are required for the use case. This shows that a blockchain that has not been designed with privacy in mind cannot be extended with arbitrary privacy-enhancing technologies without imposing significant additional costs.

V. CONCLUSION

In this paper, we presented the implementation of a tariff matching protocol from the energy domain in Ethereum. We highlighted the availability of certain cryptographic primitives in Solidity and discussed the contrasting lack of privacy by design. Even though we optimized our implementation with off-chain pre-computations and on-chain gas-reducing data handling, we found that the costs for deploying and executing the implemented smart contract are at least one order of magnitude higher than would be acceptable for the use case. This allows for the conclusion that in order to implement practical privacy-preserving smart contracts like the one in this paper, either the price for the required privacy-preserving operations in Ethereum needs to become significantly cheaper or both, the capabilities and availability of other blockchains with privacy by design have to increase significantly.

ACKNOWLEDGMENT

The financial support by the Austrian Federal Ministry of Science, Research and Economy, the Austrian National Foundation for Research, Technology and Development and the Federal State of Salzburg is gratefully acknowledged.

REFERENCES

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” *Bitcoin.org*, pp. 1–9, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] F. Tschorsch and B. Scheuermann, “Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies,” *IEEE Communications Surveys \& Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.

[3] C. Dannen, *Introducing Ethereum and Solidity*. Apress, 2017.

[4] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, “Zerocash: Decentralized Anonymous Payments from Bitcoin,” in *Proceedings – IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 459–474.

[5] G. Wood, “Ethereum: A Secure Decentralised Generalised Transaction Ledger,” Ethereum, Tech. Rep., 2017. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[6] G. Zyskind, O. Nathan, and A. S. Pentland, “Decentralizing privacy: Using blockchain to protect personal data,” in *Proceedings – 2015 IEEE Security and Privacy Workshops, SPW 2015*, 2015, pp. 180–184.

[7] G. W. Peters and E. Panayi, “Understanding Modern Banking Ledgers through Blockchain Technologies: Future of Transaction Processing and Smart Contracts on the Internet of Money,” in *Banking Beyond Banks and Money: A Guide to Banking Services in the Twenty-First Century*, T. Paolo, T. Aste, L. Pelizzon, and N. Perony, Eds. Cham: Springer International Publishing, 2016, pp. 239–278.

[8] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[9] F. Reid and M. Harrigan, "An Analysis of Anonymity in the Bitcoin System," in *Security and Privacy in Social Networks*, Y. Altshuler, Y. Elovici, A. B. Cremers, N. Aharony, and A. Pentland, Eds. Springer New York, 2013, pp. 197–223.

[10] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 839–858.

[11] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi, "Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab." in *Financial Cryptography and Data Security*. Barbados: International Financial Cryptography Association, 2016, pp. 79–94.

[12] F. Knirsch, A. Unterweger, and D. Engel, "Privacy-preserving Blockchain-based Electric Vehicle Charging with Dynamic Tariff Decisions," *Journal on Computer Science - Research and Development (CSRD)*, 2017.

[13] E. Mengelkamp, B. Notheisen, C. Beer, D. Dauer, and C. Weinhardt, "A blockchain-based smart grid: towards sustainable local energy markets," *Computer Science - Research and Development*, 2017.

[14] F. Knirsch, A. Unterweger, G. Eibl, and D. Engel, "Privacy-Preserving Smart Grid Tariff Decisions with Blockchain-Based Smart Contracts," in *Sustainable Cloud and Energy Services: Principles and Practices*, W. Rivera, Ed. Springer International Publishing, 2017, ch. 4, pp. 85–116.

[15] S. Rane, P. Boufounos, and A. Vetro, "Quantized Embeddings : An Efficient and Universal Nearest Neighbor Method for Cloud-based Image Retrieval," *SPIE Application of Image Processing*, p. 11, 2013.

[16] J. Kilian, "Founding Cryptography on Oblivious Transfer," in *ACM Symposium on Theory of Computing*. Chicago, IL, USA: ACM, 1988, pp. 20–31.

[17] A. Unterweger, F. Knirsch, G. Eibl, and D. Engel, "Privacy-preserving load profile matching for tariff decisions in smart grids," *EURASIP Journal on Information Security*, vol. 2016, no. 1, pp. 1–17, 2016.

[18] Z. Erkin, J. R. Troncoso-Pastoriza, R. L. Lagendijk, and F. Perez-Gonzalez, "Privacy-preserving data aggregation in smart metering systems: an overview," *IEEE Signal Processing Magazine*, vol. 30, no. 2, pp. 75–86, mar 2013.

[19] V. Bindschaedler, S. Rane, A. E. Brito, V. Rao, and E. Uzun, "Achieving Differential Privacy in Secure Multiparty Data Aggregation Protocols on Star Networks." in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 115–125.

[20] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *Advances in Cryptology — EUROCRYPT '99*, ser. Lecture Notes in Computer Science, J. Stern, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, vol. 1592, pp. 223–238.

[21] S. D. Rane and P. Boufounos, "Privacy-Preserving Nearest Neighbor Methods: Comparing Signals Without Revealing Them," *IEEE Signal Processing Magazine*, vol. 30, no. 2, pp. 18–28, 2013.

[22] P. T. Boufounos and S. Rane, "Efficient Coding of Signal Distances Using Universal Quantized Embeddings," *2013 Data Compression Conference (DCC)*, pp. 251–260, 2013.

APPENDIX A
DETAILED PROTOCOL DESCRIPTION

This section describes the protocol from [17] formally.

Let the set of participating utilities be denoted as $\mathcal{U}$. Each utility $u \in \{1, \ldots, |\mathcal{U}|\}$ has a list of tariffs $T_{u,l}$ with corresponding template load profiles $\mathbf{L}_{u,l}$ (denoted as set $\mathcal{L}_u$, where utilities can have different numbers of load profiles $l \in \{1, \ldots, |\mathcal{L}_u|\}$). In order to keep the template load profiles private, each utility calculates an embedding $\tilde{\mathbf{L}}_{u,l} \in \{0,1\}^m$ for each of its original template load profiles $\mathbf{L}_{u,l} \in \mathbb{R}^k$:

$$\tilde{\mathbf{L}}_{u,l} = \left\lceil \frac{\mathbf{A} \cdot \mathbf{L}_{u,l} + \mathbf{W}}{\Delta} \right\rceil \mod 2 \tag{1}$$

$\mathbf{A}$ is a random $m \times k$ matrix with i.i.d. Gaussian elements with mean 0 and variance σ^2, and $\mathbf{W}$ is a random m-dimensional vector with i.i.d. uniform elements in the range $[0, \Delta]$. Δ is both, a quantization and a security parameter, and described in detail in [21], [22]. The values of k and m are 96 and 8192, respectively.

Similar to the utilities, the customer calculates an embedding from its load profile forecast $\mathbf{F}$, denoted as $\tilde{\mathbf{F}}$.

In order to find the best matching tariff, the template load profile with the smallest normalized Hamming distance to the forecast is determined, yielding the template (once usable) load profile index l^* as well as the corresponding utility index u^*:

$$(u^*, l^*) = \operatorname*{argmin}_{u,l} ||\tilde{\mathbf{F}} - \tilde{\mathbf{L}}_{u,l}||_1. \tag{2}$$

This is possible due to the distance-preserving property of the embeddings [21], where the Euclidean distance of the original data vectors is proportional to the normalized Hamming distance of the embedded vectors with a configurable small error ϵ, i.e.,

$$||\tilde{\mathbf{F}} - \tilde{\mathbf{L}}_{u,l}||_1 \sim ||\mathbf{F} - \mathbf{L}_{u,l}||_2 + \epsilon. \tag{3}$$

The indices u^* and l^* can be shuffled in order to avoid the collection of statistics that could break privacy.

After the customer has received both indices, it can contact the corresponding utility u^* and use a one-out-of-many oblivious transfer for retrieving the actual tariff information without revealing the decision to the utility at this time [16]. Since this part of the protocol is not intended to be handled via a blockchain [14], it is not detailed here.

APPENDIX B
SAMPLE CODE

The following sample code shows an optimized version of calculating the Hamming distance required by the `evaluate` function of the implemented smart contract.

```
//Hamming distance for all 4-bit patterns.
uint8[16] map = [0, 1, 1, 2, 1, 2, 2, 3,
                 1, 2, 2, 3, 2, 3, 3, 4];

//Computes Hamming distance of two 8192-bit
//binary vectors.
function getDistance(bytes32[32] memory f,
                     bytes32[32] memory l)
  internal view returns (uint16)
{
  uint16 distance = 0;
  for (uint8 i = 0; i < 32; i++) {
    bytes32 xor = f[i] ^ l[i]; //Bit-wise XOR
    for (uint8 ii = 0; ii < 32; ii++) {
      //Process upper+lower nibble separately.
      distance += map[(uint8(xor[ii] & 0xf0))
                  >> 0x04];
      distance += map[uint8(xor[ii] & 0x0f)];
    }
  }
  return distance;
}
```