Rass *et al. EURASIP Journal on Information Security* (2015) 2015:7
DOI 10.1186/s13635-015-0025-9

EURASIP Journal on Information Security
a SpringerOpen Journal

RESEARCH Open Access

Private function evaluation by local two-party computation

Stefan Rass*, Peter Schartner and Monika Brodbeck

Abstract
Information processing services are becoming increasingly pervasive, such as is demonstrated by the Internet of Things or smart grids. Given the importance that these services have reached in our daily life, the demand for security and privacy in the data processing appears equally large. Preserving the privacy of data during its processing is a challenging issue that has led to ingenious new cryptographic solutions, such as fully homomorphic encryption (to name only one). An optimal cryptographic support for private data processing must in any case be scalable and lightweight. To this end, we discuss the application of standard (off-the-shelf) cryptography to enable the computation of any function under permanent disguise (encryption). Using a local form of multiparty computation (essentially in a non-distributed fashion), we show how to execute any data processing algorithm in complete privacy. Our solution can, for example, be used with smart grid equipment, when small hardware security modules are locally available (such as in smart meters).

Keywords: Private function evaluation, Security, Homomorphic encryption, Multiparty computation, Secure function evaluation

1 Introduction

Smart metering, Internet of Things, and Internet applications increasingly require data distribution over a very large scale—often spanning across nations and in some cases continents. In many of these applications, the dissemination of data could be made more efficient and effective by ensuring that only relevant data is delivered to interested consumers by taking into account the information content of some of its meta-information, such as the location. Other applications like smart grids may do data concentration at several (remote) points in the network (data concentrators) to compile data in a fully encrypted form for further processing (and decryption) at the head-end system. However, for applications that are not willing to reveal information or meta-information to "un-trusted" services (clouds, etc.), the infrastructure has no alternative but to rely on functional (e.g., homomorphic) encryption for secure data aggregation or dissemination without disclosure.

In this work, we discuss an application of standard (homomorphic) encryption to the problem of data processing in privacy. Especially, we shall avoid the use of non-standard (and perhaps heavy-weight) primitives such as fully homomorphic encryption or similar. Instead, we are after a generic approach to secure function evaluation that works on entirely classical building blocks that are already widely available and implemented. Before that, however, let us briefly fit our proposal into the existing related scientific landscape.

2 Secure function evaluation: the three cryptographic ways

Encryption is normally designed to prevent any meaningful processing of data. A few systems, like RSA or ElGamal encryption (that both work on groups), allow a single operation to be applied to the ciphertext but propagate through to the plaintext. We call this a *group-homomorphic encryption*, and it is the simplest form of data processing under disguise. Fully homomorphic encryption (FHE) advances over such group homomorphism in allowing ≥ 2 different operations to be applied to ciphertext so as to effectively modify the hidden plaintext. Originating from Gentry's seminal first FHE scheme [1], most subsequent schemes (e.g., [2–5] to name a few) allow for an unlimited number of additions and multiplications on the ciphertext. In any case, the goal is to have

*Correspondence: stefan.rass@aau.at
Universität Klagenfurt, Universitätsstrasse 65-67, Klagenfurt, AT, Austria

a (reasonably powerful) set of basic operations that enable us to construct arbitrarily complex functions that work on ciphertexts and rely on homomorphy to let the externally applied action penetrate the encryption to modify the plaintext accordingly.

The computational model to describe the functionality is in most cases a boolean or arithmetic circuit family $(C_n)_{n\in\mathbb{N}}$, where C_n is a circuit composed from boolean or arithmetic gates that processes inputs of a fixed length n (usually n bits). For every such circuit C_n describing a function $f : \{0,1\}^n \to \{0,1\}^m$, a respective *evaluation circuit* can be compiled that processes an encrypted input x of length n into a ciphertext containing $f(x)$ by virtue of FHE. An alternative and older concept due to A. Yao [6] proposes to represent signals in the circuit by ciphertexts and to emulate the functionality of a (logical or arithmetic) gate by a "lookup table." These *garbled circuits* (GC) require a secure specification of signal representations and the transformation from the plain circuit functionality to the garbled circuit that works on the encrypted signal representations. In any case, and unlike FHE, GC thus involve at least two parties in every computation. It has nevertheless been recognized as a powerful and promising alternative, with many theoretical and practical achievements [7], and powerful experimental implementations [8–11]. The last reference here is most closely related to our work, since it also presents a scheme to execute assembly code, based on garbled circuits.

The idea of involving multiple instances towards secure computation of functions is itself a third branch of research related to private function evaluation, known as *multiparty computation* (MPC) [12]. We leave this third alternative aside here for space reasons and for the sake of describing a fourth and entirely generic construction, which is based on standard encryption (unlike FHE) and works without interaction over distributed entities (as would be the case for GC [13] and MPC), which imposes overhead by additional network traffic.

3 A fourth route

Going away from circuits to represent a functionality, Turing machines offer another very handy model for an(y) algorithm. Unlike the basic operations in arithmetic/logical circuits (additions, multiplications, etc. that call for the respective homomorphy in the encryption), Turing machines can accomplish their basic operations much simpler and by exploiting group homomorphy only. This observation has led to the proposal of running Turing machines on encrypted data [14, 15]. The latter reference—theoretically—proves that running Turing machines on encrypted data does not even call for enhanced encryption and can be done using a generic extension to a standard homomorphic encryption function. Alas, Turing machines are not suitable for practical purposes, and putting the concept to practice by switching to a more convenient representation like assembly code induces severe vulnerabilities in the overall concept. These will be in the center of interest in the following, along with respective solutions.

The upcoming description of problems will be based on the following condensed description of how to execute encrypted assembly code. For that matter, let $E : (\mathbb{G}_m, *) \to (\mathbb{G}_c, \cdot)$ be a probabilistic encryption that converts a plaintext $m \in \mathbb{G}_m$ into a ciphertext $c \in \mathbb{G}_c$. Assume that E is homomorphic in the sense that under a key k, we have $E(m_1 * m_2, k) = E(m_1, k) \cdot E(m_2, k)$, and that E is probabilistic in the sense that $E(m_1, k)$ is not distinguishable from $E(m_2, k)$ whenever $m_1 \neq m_2$ and both plaintexts have equal length, denoted as $|m_1| = |m_2|$.

The main tool to prove the computability of any function by a Turing machine, whose tape is encrypted, is a homomorphic public-key encryption with *plaintext equality testing* (HPKEET). Ad hoc constructions have been given in [16], and a generic construction is found in [15]. In general, a HPKEET scheme adds two algorithms *Aut* (to authorize comparisons) and *Com* (to perform comparisons) to a conventional encryption scheme composed from key generation, encryption, and decryption algorithms. Invoking $k_{com} \leftarrow Aut(sk)$ upon input of a secret key sk creates a comparison key k_{com}. This one can be used to compare two ciphertexts c_1, c_2 for equality of the inner plaintexts. That is, letting $D(\cdot, sk)$ denote the decryption of the input under the secret key sk, equality is indicated upon $Com(c_1, c_2, k_{com}) = 1 \iff D(c_1, sk) = D(c_2, sk)$. The crucial point here is that the comparison key cannot be used to disclose any plaintext as such.

Given a HPKEET scheme, it is a simple matter to define the actions of a Turingmachine on an encrypted tape. Essentially, its actions are a matter of manipulating encrypted tape symbols (by exploiting the encryption's homomorphy) and looking up the proper state transition by virtue of the plaintext comparison facility (Aut, Com). Likewise, it is equally simple to use (Aut, Com) to build a lookup table in which the result of an arithmetic or logical operation can be obtained from two encrypted operands $c_1 = E(op_1, pk), c_2 = E(op_2, pk)$. Note that these operands are both encrypted under the same public key pk, as they both originate from the data provided by Alice. The public key pk can thus be either Alice's own key (if she simply outsources a computation to use the results for her own purposes) or the public key of another end-user, e.g., this would be the head-end in a smart metering application, where "Alice" is the smart meter that sends the measurements. In any case, the user Alice can authorize an external party, say

a distrusted *processor*, to compute something on Alice's behalf, given only encrypted data and the comparison key $k_{com} \leftarrow Aut(sk)$ that enables manipulations on encrypted operands $E(op_1, pk), E(op_2, pk)$. We stress, however, that the computing entity (as being regarded not trustworthy) is never the party that generates any keys. Its purpose is exclusively the processing of data, which—at the bottom and per instruction—is simply a lookup of (op_1, op_2) in some operation's lookup table by calling the comparison algorithm *Com* to locate the row (marked by op_1) and column (marked by op_2). Thanks to the HPKEET scheme, we can avoid decrypting the operands and obtain the result of the secret operation directly from the table. Indeed, the HPKEET in that sense re-creates the lookup tables that are at the core of GC, with the important difference that HPKEET-based lookup tables are reusable for all operations of the same type (a property that has only recently been studied for garbled circuits [17]).

The appeal of this construction lies in the easy possibility to realize *any* operation (addition, multiplication, Boolean operators, etc.) simply by constructing the proper lookup table. Such flexibility is rarely found in other schemes. However, on the downside, it also defeats the security of the underlying encryption, as we discuss next.

3.1 Encrypt-and-compare attacks: vulnerabilities against passive adversaries

A careless use of HPKEET is generally insecure in various ways: first, plaintext comparisons enable the processor to recognize identical plaintexts that reoccur during a longer computation. Ultimately, e.g., if the computation is on single bits (such as is the case for many FHE schemes), the processor would be left only two possibilities for the hidden plaintext. This is because all equal bits are recognizable as being identical, and the remaining uncertainty is only about which is the "zero" and which is the "one" bit. This problem is not existing in GC, since all gates use different representations.[1] For similar reasons, the problem is also not relevant for FHE, since the evaluation there is based on circuits and does not require equality checks on plaintexts.

More generally, if a plaintext is encrypted under a HPKEET scheme and the plaintext space is feasibly small, then a brute-force search by trial encryption-and-comparison is possible: the adversary can simply encrypt a candidate plaintext under the known public key and compare it to the unknown plaintext by invoking *Com*. Normally, this attack can be thwarted by either imposing the requirement of high min-entropy of the plaintext [18] or by prescribing a secret encoding of the (few) plaintexts, so that the adversary no longer knows which plaintexts to test [15]. The latter fix basically concerts the public-key scheme into a symmetric one.

3.2 The chosen instruction attack: vulnerabilities against active adversaries

Imagine a computing platform that supports a real-life assembly language interpretation, say MIPS assembly code [19]. Furthermore, let the assembly interpreter execute an instruction, say `add r1, r2, r3` (with the semantic that `r1` $\leftarrow$ `r2` $+$ `r3`) by a lookup table based on any encryption (e.g., HPKEET) scheme. Finally, assume that the instruction set contains operations for arithmetic manipulations, especially including subtractions and divisions.

Under these hypotheses, the previously described trial encryption-and-comparison discovery of an unknown plaintext is again possible, irrespectively of high min-entropy or any secret encoding of the plaintexts (thus, any transformation to increase the uncertainty would be non-effective).

The attack works as follows: let the register `r0` contain an(y) unknown value. The attacker first encryptedly computes the zero value $0 =$ `r0` $-$ `r0` by submitting `sub r1, r0, r0`. Then, the HPKEET equality testing can be used to locate any register with a nonzero content, let us call it `r2` (note that this requires the comparison ability, which the attacker has gained by receiving the authorization token k_{com}, and from the fact that all operands are encrypted with the same public key (thus, k_{com} allows to compare any pair of operands; cf. also Fig. 3). From `r2`, we can compute (an encrypted and possibly encoded version of) the values 1 and 0 by doing a division with remainder. In MIPS assembler, the command would be `div r2, r2`, which leaves the quotient $1 =$ `r2/r2` and remainder $0 =$ `r2 MOD r2` in some special-purpose registers. Once this is accomplished, it is a simple matter to enumerate the values $2, 3, 4, \ldots$ by repeated execution of `add`-instructions. The so-constructed dictionary of candidate plaintexts can then be used with the HPKEET plaintext comparison facility to disclose all encrypted unknown plaintexts. The dictionary (i.e., plaintext space) is necessarily very small, since processing ℓ-bit words requires lookup tables of size $O(2^\ell) \times O(2^\ell) = O(2^{2\ell})$ bits.[2] Thus, to remain feasible, we are bound to processing chunks of up to 8 bits (and hardly more). While this is a good advantage over other schemes that work on single bit level (as do some FHE schemes and garbled circuits), it nevertheless enables a quick brute-force test for equality of a given encrypted plaintext against the so-constructed dictionary.

The crucial point here is all this being doable without knowledge of any secret key or secret encoding. Thus, we may intermediately conclude that private function evaluation using homomorphic encryption with equality checking (i.e., HPKEET) is inevitably insecure in the *single-party* setting. Our next step is fixing these problems by switching to a local two-party setting.

4 Secure function evaluation in a local two-party setting

Both of the previous attacks make frequent use of the plaintext comparison facility that the HPKEET scheme supplies. Since this is equally important to construct the lookup tables, we must somehow remove the comparison ability from the attacker, while retaining the ability to do the lookups. A simple solution is to put the lookup into a particularly secured hardware environment which then acts as a local but separated second party in the computation.[3] The concrete implementation of the secured hardware environment depends on two factors: the required security level (which in turn depends on the security level of the processed data) and the amount of data (and hence the number of operations per second) that has to be processed. So in some application scenarios, a crypto-smart card would perfectly fit the requirements and in others, a "full" hardware security module (HSM) is needed. Obviously, this influences the costs for such a device as well. Throughout this article, we will use the term HSM, whenever referring to a specifically secured hardware environment. For security, we assume that secrets being stored in the HSM are neither physically nor logically accessible (either directly or indirectly, say through side-channels). Note that the functionality assumed within the HSM covers access control (to prevent the HSM from misuse as a comparison oracle) and doing table lookups (which amounts to only simple cryptographic operations). Thus, the HSM itself can be quite lightweight (thus cheap, e.g., a crypto-smart card).

That is, each instruction of the form `opcode r1, r2, r3` is submitted to the HSM, which performs the operation using the lookup table addressed by the `opcode`. Additionally, if the encryption is probabilistic, the HSM can easily re-randomize the output to make identical outputs indistinguishable (a simple technique for ElGamal encryption is multiplying a ciphertext with an encryption of the unit element, which changes the randomizer in the ciphertext but leaves the inner plaintext unchanged). Figure 1 illustrates the process, with particular indication of points in the flow when the data becomes distinguishable; these are the times when the chosen instruction attack could be mounted, so this is where the HSM's protection becomes crucial.

The expressive power of the register machine (equivalently the Turing machine) model is retained, while neither of the above attacks work anymore. Moreover, if the HPKEET is constructed from a conventional semantically secure cipher, say ElGamal encryption (such as described in [15]), then the distrusted processor sees only a sequence of ciphertexts, from which nothing useful can be obtained (based on the intractability assumption that underlies the encryption) by standard arguments [20]. More precisely, the security of our generic construction (in terms of data item indistinguishability) follows directly from the semantic security of the underlying encryption, since the aforementioned chosen instruction attack no longer applies by virtue of the HSM. Thus, we can conclude that this kind of *local two-party* computation is flexible in the sense of admitting the execution of arbitrary assembler instructions and inherits its security from the underlying encryption (which by use of the HSM is no longer a HPKEET but a normal public-key encryption).

4.1 Security problems by code execution patterns

It must be stressed that the above security argument applies only to arithmetic and logical instructions but does not cover memory addressing or branching. Both of these, unfortunately, may leak lots of information through observations of the program execution patterns. To illustrate the problem, imagine a division being done (encryptedly) by repeated subtractions. In that case, counting the number of iterations of the loop immediately reveals the result (even if the plaintext comparison facility is concealed within the HSM). Likewise, memory addressing needs (probabilistically) encrypted addresses to be translated back to (deterministic) physical memory addresses. A plain conversion is not advisable, as in this case, we could compare two encrypted values simply by converting their contents to memory addresses for checking their equality. Similarly, conditional branching may be exploited to construct a plaintext comparison function as follows: MIPS assembler, for example, supports branching upon equality (`beq`). Observing which case is being taken ("then" or "else") then immediately tells the observer if the register contents have been equal or not. At this point, the attacker can mount a trial encrypt-and-compare attack again by a chosen instruction attack involving a sequence of `beq`-instructions.

4.2 Fixing memory access and branching leakages

Protection from information leakage from memory access patterns is a long recognized issue that is solved by cryptographic techniques like oblivious RAM [21–25] and private information retrieval (PIR) [26–29]. Both can be used in a black-box manner to resolve the above issues, yet the concrete selection of a scheme and analysis of security and complexity impacts is beyond the scope of this article (subject of future research). Perhaps even more difficult is the protection against misuse of branching instructions. Here, proper code obfuscation, in particular the hiding of the "then" and "else" branches of instructions appears necessary. Note that this intentional security from code obscurity is in this particular case a necessity to thwart chosen instruction attacks by stripping the semantics from the instructions (much like encryption removes the meaning from a plaintext by casting it into a ciphertext). This is perhaps the most challenging issue and an

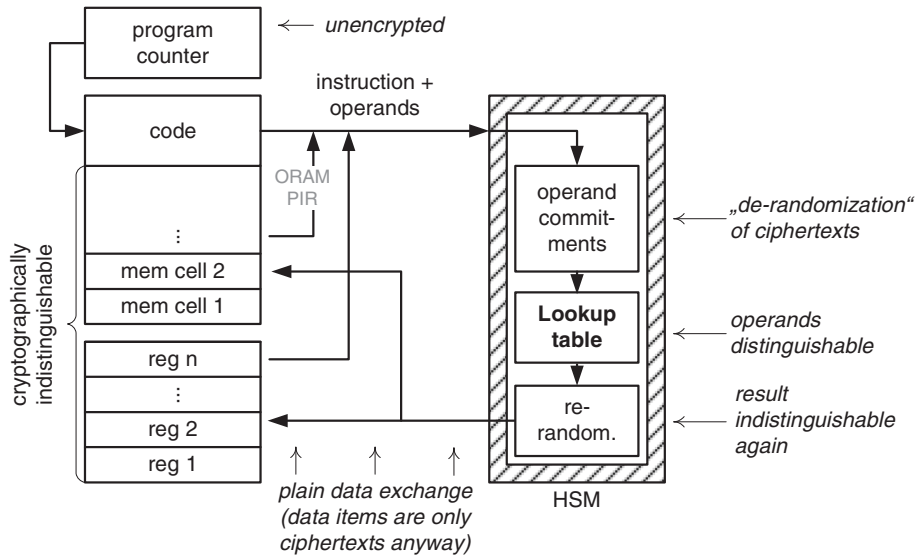

Fig. 1 Execution flow of a single instruction

interesting direction of future research, as a deeper treatment is beyond the scope of this article. We close the discussion by remarking that this attack is not specific for our proposal or any encryption and must be carefully analyzed for other computing platforms or general approaches, such as [11], as well.

Summarizing our proposal so far, it appears as a not too difficult matter of combining secure hardware with standard cryptographic primitives into a platform that supports the execution of assembly code on encrypted data. The crux here is simply to execute assembly code by lookup tables, while preventing the potentially hostile execution environment from constructing a plaintext comparison functionality on its own to do a trial encrypt-and-compare ciphertext disclosure. The hardware security module in that case must only handle table lookups; it can be implemented as its own module or also integrated into an existing hardware protected module or area (thus adding only little additional computational burden to the HSM already in use for other purposes). Especially, this makes it much more lightweight than a general-purpose processor that decrypts operands, executes the instructions, and re-encrypts the result again (the operations for a plaintext comparison are much more efficient than this approach and allow for a much thinner HSM architecture). A hypothetical such architecture is sketched in Fig. 2. Implementing and testing it is another aspect of future work to be done.

5 Application in the smart meter networks

Coming to the concluding application, smart grid and smart meter networks appear as natural candidates for the implementation of our proposed computing architecture. Herein, let us assume that a smart meter (or general sensor network) is composed from sensors (smart meters, etc.) and data concentrators in a ring/tree (mixed) network topology. Within the data concentrator, it appears as not too restrictive to make a HSM available to do secure local two-party computation such as sketched above. Under this hypothesis, the data concentrator can go ahead and process encrypted data retrieved from the smart meters (sensors) without decrypting anything and, more importantly, without privacy breaches and under end-to-end encryption spanning much wider ranges than the smart meter network. Figure 3 displays an example snapshot of such a possible application. Here, the entire process depicted in Fig. 1, and the computing platform (Fig. 2) is entirely encapsulated within the data concentrator (DC), that is, the DC hosts a HSM and memory to execute the instructions for the data aggregation algorithm, as described before.

To be more concrete on the privacy issue here, first recall that the entire construction is generic and as such inherits its cryptographic assumptions from the underlying encryption (e.g., it is based on discrete logarithms if ElGamal encryption is used or based on factorization or quadratic residues if Paillier encryption is employed). As for privacy in light of collusions among the computational parties in the system (say, the head-end collaborating with the data concentrator), the *locality* of our two-party approach comes into play: notice that, unlike the term "two-party" would indicate, no collaboration between any two computing instances in the system will reveal anything, since the actual two-party computation involves a computing instance and its local HSM as the two players. Thus, unless the data concentrator, for example, can

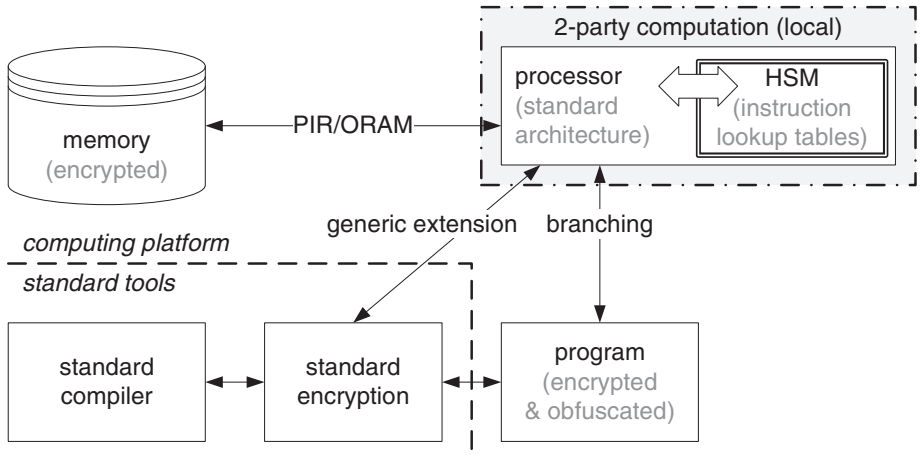

Fig. 2 Computing platform sketch

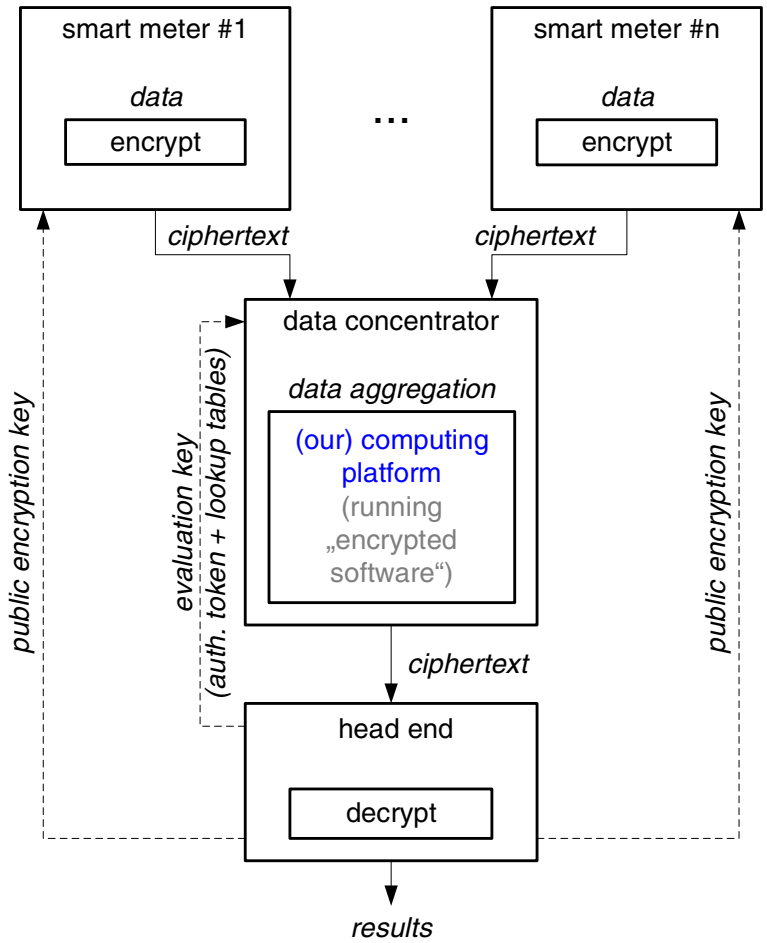

Fig. 3 Application in smart meter networks

"collude" with its HSM (which is ruled out by the definition of a HSM, which acts autonomously and should resist any compromission attempts), no privacy breach can occur. Conversely, if the DC and the head-end join forces, then neither can gain anything beyond what is known already, since there is no multiparty computation done between these instances.

6 Implementation and evaluation

The system has been implemented in a Java prototype, in which the HSM has been emulated by a designated class with the proper visibility modifiers (i.e., private data within the HSM is not accessible from outside). The prototype understands a dialect of MIPS assembly code [19] and can emulate operations on 8, 16, 32 bit …words by processing chunks of 4 bits using its internal lookup tables. The lookup tables themselves are organized as conventional hashtables that are keyed on the "derandomized" versions of the input ciphertexts.[4] The complexity of a single lookup is thus approximately $O(1)$ (up to probing inside the hashing). Code obfuscation has been implemented in a very simple fashion by choosing a random set of pseudonyms for the branching instructions, so that branches upon equality or less than or greater than relations are indistinguishable between different encryptions of the same code. The memory access has been left deterministic (adding oblivious RAM or PIR is a matter of future work), so the timing estimates following in Table 1 exclusively relate to arithmetic instructions.[5]

6.1 Benchmarks

A practical limitation arises from the exponential growth of lookup tables in terms of the block size that we can process per operation. That is, if the processor is supposed to work in b bit words, then the lookup table would consist of $2^b \times 2^b = 2^{2b}$ ciphertexts. To keep things practical, we must thus emulate larger register sizes by operations on smaller blocks, say 4-bit chunks. For example, an addition of two 64-bit registers would thus be done by adding 16 blocks within the register individually. This keeps the evaluation key at a reasonable size ($16 \times 16 = 256$ ciphertexts per instruction and about 10 different instructions in our dialect[6]), at the cost of taking more time to execute the assembly instruction. Table 1 shows some benchmark results[7] for our Java prototype. The given benchmarks are the average time estimates (in seconds) taken over 1000 repetitions of each operation using random operand values. The times exclude the overheads for encryption and decryption of inputs and results and also do not include the time for loading the virtual machine or the respective keys. The platform has been an Ubuntu 14.04 LTS (64 bit), running on an AMD A8-4500M APU with Radeon HD Graphics at 1.9 GHz, and with 8 GB RAM. The cryptographic keys used in the experiment had 2048 bit (adhering to nowadays recommendations for modular arithmetic).

Table 1 Benchmark figures for Java prototype implementation (some arithmetic/logical instructions; average execution time (in seconds) for 1000 repetitions of the instruction on random operands)

Register size	And	Add	Sub	Mult
16 bit	0,942	2,592	3,666	48,924
32 bit	2,406	7,122	10,224	≈ 216

As the execution times indicate, there is a strong need for optimization,[8] especially in terms of parallelization. A closer inspection of the scheme in [15], however, shows that using parallelization, the overall effort for processing a single block, i.e., a table lookup, can be done using 1 modular exponentiation + 2 modular multiplications + 1 modular inversion + $O(1)$ for the hashtable query.

It may be well the case that these execution times are outperformed by specially designed aggregation protocols, which work much more efficiently at the cost of computing only a fixed aggregation function. In smart grid appplications, however, the data concentrators may require much more intelligence beyond simple data aggregation, especially in cases where decentralized load balancing is required. In general, the logic of the utility network control layer (the supervisory control and data acquisition (SCADA) system) can, using our platform, be integrated in a way that respects the client's privacy, while still being able to do all the necessary processing to effectively manage the smart grid. In such future utility network architectures, hand-crafting fast aggregation and control algorithms may quickly become infeasible in practice. This is where a general-purpose computing platform, in combination with a standard compilation chain may unleash its power.

6.2 Computational complexity

Going into that direction, we can reconsider our approach relative to other cryptographic techniques like FHE. While a qualitative comparison to FHE is not entirely meaningful (since the latter is not a multiparty primitive unlike our proposal), a comparison in terms of complexity indeed provides some insights. Let us consider some fast FHE and GC platforms for our comparative discussion. Assuming that all the operations (related to FHE, GC, or our scheme) are implemented under the same optimizations and on the same platform, we may restrict our comparison to the (asymptotic) complexities[9] that count how many bit-operations are required to homomorphically evaluate a given function, represented as an arithmetic circuit (in case of FHE and GC) or an

assembly program (for our scheme). For the cryptographic operations, let $t \in \mathbb{N}$ denote the security parameter. Reference [30] notes $\tilde{O}(t^{3.5})$ operations per arithmetic gate, contrary to [2], who achieve $\tilde{O}(t)$ operations for multiplication (and hence also addition) gates. The downside of the latter scheme is, however, the encryption taking $\tilde{O}(t^{10})$ many steps. Let us compare this to our setting now in terms of complexity: since our scheme is generically based on standard encryption, its bit-complexity is proportional to that of the underlying scheme; thus $\tilde{O}(t)$ if we use ElGamal encryption, for example. As for homomorphic evaluation, parallelization reduces the overhead for a single lookup operation to $\tilde{O}(t)$ bit-operations (assuming that exponentiation is the most expensive task herein). Processing two registers (e.g., multiplying them) of fixed size n bits, then takes $\tilde{O}(c(n) \cdot t)$ bit-operations, if the operation on the registers takes $c(n)$ steps (for a multiplication, this would be $c(n) \in O(n^2)$ using a non-optimized standard pen-and-paper method). In terms of our complexities, since n is constant, the complexity of an instruction execution is thus still $\tilde{O}(t)$ bit-operations (with the value of n scaling the constant implied by the $\tilde{O}$). The concrete effort of course depends on the particular operation, with multiplication and division being quite expensive examples. However, FHE and GC both require circuits to do the arithmetic; therefore, their complexity is as well given by $c(n) \cdot \tilde{O}(t^{\alpha})$, where $c(n)$ is the size of the circuit to process n-bit inputs, and α depends on the chosen scheme ($\alpha = 3.5$ for [30] or $\alpha = 1$ for [2]). In fact, since our scheme can process more than single bits in one blow, the complexity $\tilde{O}(c(n) \cdot t)$ must further be cut down by a factor of 4, if we work on 4-bit chunks per lookup, for example (this is another difference and advantage over circuit based evaluation as done in FHE or GC). In any case, it is equally efficient as [2] on homomorphic evaluation but much faster than this scheme on encryption and decryption. Thus, despite the timing results being somewhat blurred by overheads induced from Java's virtual machine, and the inefficient implementation, a designated and optimized platform (like the ones that exist for GC [7] and FHE) may redraw the picture.

Another qualitative comparison relates to the ability to process inputs of arbitrary size: both FHE and GC require freshly compiled circuits to process inputs of growing lengths. MPC, on the other hand, can work with any length but needs to compile the function evaluation into an interactive protocol, which induces network traffic. By keeping the interaction *local* in our scheme, we reduce the communication overhead to a local communication on the motherboard, thus gaining efficiency from good hardware implementations. Using a hardware security module herein addresses some (but not all) of the security issues, and enhancing the proposed computing architecture by ORAM/PIR and code obfuscation to gain further security, is a matter of future research.

6.3 Security

Semantic security (i.e., the inability of a polynomial time-bounded adversary to extract any useful information from a given ciphertext), follows easily for *passive* adversaries: observe that thanks to the HSM, the adversary cannot do any plaintext equality checks along a cryptanalysis, so everything that can be harvested from outside the HSM is essentially a bunch of ciphertexts. Since semantic security of a single public-key encryption extends to polynomially many such encryptions (multi-message security [31]), our scheme is semantically secure against passive attackers, provided that (1) the adversary is polynomially time-bounded (in the security parameter that the encryption uses) and (2) the time-complexity of the algorithm being executed is at most polynomial (to retain the known proofs of multi-message semantic security applicable). The latter requirement induces a subtle issue in the choice of the security parameter of the encryption, namely that the problem size (that determines the time-complexity of the algorithm) must be polynomially related to the security parameter (that limits how many ciphertexts can be emitted before the scheme becomes insecure). Roughly speaking, the security parameter of the encryption should be set at least proportional (or larger, say quadratically dependent) on the problem size. For example, if n data items are expected to be processed, then the security parameter should be set to $t \approx n$; the point here is that this parameter usually determines the bitsize of the encryption keys and the size of the underlying algebraic structures. In case of ElGamal encryption, the parameter t is the bitsize of the finite group that we work in. If this is bitsize t and we are working in standard modular arithmetic, then the group is roughly of size 2^t.[10] So, if t is dependent on the problem size n and the time-complexity is polynomial, say $p(n)$ steps to process n items, then the execution emits only $p(n) \approx p(t)$ many ciphertexts, i.e., polynomially many in the security parameter t. Under this setting, the scheme remains secure. Practically, if we expect, say $n = 10^9$ measurements to be processed within a data concentrator, then the security parameter, i.e., bitsize, can be set to $t \geq \log_2(n) = \log_2(10^9) \geq 60$ bit. That is, for processing a billion measurements, nowadays recommended key sizes (at least 320 bit for elliptic curve cryptography) are more than sufficient to keep our data secure.

The aforementioned vulnerabilities against *active* attacks were originally discovered along experiments with the prototype, which led to continuous improvements of the theory and the prototype as such. Details and further findings will be reported in a follow-up work. An observation from these experiments is the potential relevance

of the aforementioned attacks in other contexts such as FHE, GC, or computing platforms (e.g., [11]). Indeed, it is a prescribed standard procedure to construct circuits in FHE or GC so that each data flow path is equally likely, in order to avoid the attacker learning the data from the data flow; thus FHE and GC suffer from similar (yet not entirely identical) issues as our scheme. Therefore, it does not seem that attacks like the described ones are per se absent in alternative schemes. While FHE or GC evaluation circuits do not offer the attacker the flexibility of submitting instructions of her/his own choice (thus, the CIA-attack is not immediately relevant in FHE or GC), the structure of a circuit (whether garbled or not), may nevertheless leak information about the underlying data, since it is compiled from it (the notion of *circuit privacy* is known, yet a conventional garbling procedure primarily hides the functionality of gates but not the structure of the circuit itself).

Note that all this applies only under the (initial) assumption that the HSM is entirely tamper-proof. If it somehow leaks out its secret, i.e., the comparison token k_{com}, then the previously described vulnerabilities are fully applicable again.

The appeal of a computational platform that admits the processing of encrypted data and code lies in its compatibility with standard compilation chains. That is, taking MIPS assembly a last time as our example, the code itself needs no change except for the fact that its registers contain ciphertexts rather than plain integers (changes to the code mostly apply to constants encoded therein, which must be encrypted for compatibility with encrypted memory and register contents). In any case, however, we can use a standard compiler to create the processing software and can "directly" run this code on encrypted data. Note that this is an essential difference to alternative approaches (like GC or FHE), which require special-purpose compilers [32–34] and designated code/circuit generation [35] that is specific for a particular function. Unlike this, the computing platform sketched here is "general purpose".

7 Conclusions

This work presents a general-purpose framework and computing architecture for data aggregation, with possible applications in smart grids or sensor networks (although the scheme is not restricted to any of these areas). Our proposal is meant for use in situations where data aggregation cannot be tied or tailored to the specific application, and future extensions of the aggregation are somewhat foreseeable. In practical implementations, a general-purpose data processor may be outperformed by special-purpose aggregation protocols, which can be tailored to the specific aggregation function. This performance gain is bought at the cost of fixed functionality (i.e., the aggregation can deliver exactly a defined output). Thus, when future new requirements or extensions to the aggregation are foreseeable, a more powerful computing platform may become necessary.

The main idea presented in this work is that of using a lightweight hardware protection for simple table lookups, to implement a simple secure processor for conventional assembly code (the entire platform is not lightweight in computational terms, but no entirely cryptographic solution would be). To this end, we sketched an architecture to implement a fully functional data processing utility that executes arbitrary programs under permanent disguise of an encryption. Our scheme is generic, in the sense of being usable with standard encryption like ElGamal's scheme. Along practical experiments with an implementation of the idea, we identified several vulnerabilities to break the encryption (despite the encryption's proven security against the standard attack patterns), by which what we call a *chosen instruction attack*. We described various potential countermeasures related to this attack and demonstrated the diversity of the required protection beyond pure cryptography (such as code obfuscation, branch protection, and secure memory access being necessary). Nevertheless, the general possibility of arbitrary data processing in privacy (under encryption) has been verified experimentally in software simulations, and practical experiments on physical hardware are a natural next step.

Endnotes

[1] The circuit description has a size proportional to the number of gates, with the factor being a multiple of the cipher's blocklength.

[2] Each table entry being an HPKEET ciphertext. For the generic construction in [15], for example, the ciphertext size is proportional to the size of the underlying ElGamal ciphertext.

[3] A purely cryptographic approach to lookups *without* comparisons is described in [36], but this technique applies only to one-dimensional tables so far.

[4] Roughly speaking, the authorization (comparison) token k_{com} lets us strip the randomizer from the ciphertext, so that the scheme becomes deterministic. Thus, identical derandomized ciphertexts imply identical inner plaintexts (see [15] for full details).

[5] In a real-life implementation, the timing is expectedly dominated by the communication with the HSM, which may be the bottleneck point for performance.

[6] Remember that MIPS is a RISC architecture.

[7] Notice that a comparison with other schemes for private function evaluation is not very expressive yet, since the total overhead including additional (and yet unclear) precautions to protect the flow control and memory access pattern leakages would add an unknown

lot to the overhead. Nevertheless, a comparison of the efficiency of blind Turingmachines relative to competing approaches has been done by [37].

[8]For example, the Java prototype uses a humble and wasteful textual encoding of numbers, thus induces a huge lot of potentially avoidable string parsing overhead.

[9]Here denoted in terms of $\tilde{O}(f)$, which is a shorthand for $O(f(\log f)^k)$ for some fixed integer k (implied by $\tilde{O}$ in addition to the other constants).

[10]The story is far more involved for elliptic curve groups, but let us keep the example simple here.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
SR proposed the original concept of a blind Turing machine that motivated this work and built the first prototype version with security against passive adversaries. PS recognized the information leakage from the branching instructions and contributed the solution by hardware security support. MB identified weaknesses against active attacks during her further development of the prototype and pointed out some of the solutions mentioned here. All authors read and approved the final manuscript.

Acknowledgements
We wish to thank the anonymous reviewers for the valuable suggestions, which provided much of an improvement for the text and also delivered interesting ideas for subsequent research.

Received: 2 July 2015 Accepted: 27 November 2015
Published online: 22 December 2015

References

1. C Gentry, in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*. STOC '09. Fully homomorphic encryption using ideal lattices (ACM, New York, NY, USA, 2009), pp. 169–178. doi:10.1145/1536414.1536440
2. M van Dijk, C Gentry, S Halevi, V Vaikuntanathan, in *EUROCRYPT*. LNCS. Fully homomorphic encryption over the integers, vol. 6110 (Springer, Berlin Heidelberg, 2010), pp. 24–43
3. NP Smart, F Vercauteren, in *PKC*. LNCS. Fully homomorphic encryption with relatively small key and ciphertext sizes, vol. 6056 (Springer, Berlin Heidelberg, 2010), pp. 420–443
4. Z Brakerski, V Vaikuntanathan, in *FOCS 2011*. Efficient fully homomorphic encryption from s LWE (IEEE, Berlin Heidelberg, 2011), pp. 97–106
5. Z Brakerski, C Gentry, V Vaikuntanathan, in *ITCS*. (Leveled) Fully homomorphic encryption without bootstrapping (ACM, Berlin Heidelberg, 2012), pp. 309–325
6. AC-C Yao, in *FOCS*. How to generate and exchange secrets (extended abstract) (IEEE, Berlin Heidelberg, 1986), pp. 162–167
7. Y Huang, D Evans, J Katz, L Malka, in *20th USENIX Security Symposium*. Faster secure two-party computation using garbled circuits (USENIX Association, Berlin Heidelberg, 2011)
8. K Järvinen, V Kolesnikov, A-R Sadeghi, T Schneider, in *Cryptographic Hardware and Embedded Systems, CHES 2010*. Lecture Notes in Computer Science, ed. by S Mangard, F-X Standaert. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs, vol. 6225 (Springer, 2010), pp. 383–397. doi:10.1007/978-3-642-15031-9_26
9. S Zahur, D Evans, in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Circuit structures for improving efficiency of security and privacy tools (IEEE Computer Society, Washington, DC, USA, 2013), pp. 493–507. doi:10.1109/SP.2013.40
10. EM Songhori, SU Hussain, A-R Sadeghi, T Schneider, F Koushanfar, in *36th IEEE Symposium on Security and Privacy*. Tinygarble: Highly compressed and scalable sequential garbled circuits (Oakland, San Jose, California, 2015)
11. XS Wang, C Liu, K Nayak, Y Huang, E Shi, in *IEEE Symposium on Security and Privacy (S & P)*. Oblivm: A programming framework for secure computation, (2015). http://www.cs.umd.edu/~elaine/docs/oblivm.pdf, accessed 07 December 2015
12. M Hirt, U Maurer, Player simulation and general adversary structures in perfect multiparty computation. J. Cryptol. **13**, 31–60 (2000). doi:10.1007/s001459910003
13. T Schneider, M Zohner, in *Financial Cryptography and Data Security*. Lecture Notes in Computer Science, ed. by A-R Sadeghi. Gmw vs. yao? efficient secure two-party computation with low depth circuits, vol. 7859 (Springer, 2013), pp. 275–292. doi:10.1007/978-3-642-39884-1_23
14. S Goldwasser, Y Kalai, RA Popa, V Vaikuntanathan, N Zeldovich, How to run Turing machines on encrypted data. Cryptology ePrint Archive, Report 2013/229 (2013). http://eprint.iacr.org/, Accessed 07 Dec 2015
15. S Rass, Blind Turing-machines: Arbitrary private computations from group homomorphic encryption. Int. J. Adv. Comput. Sci. Appl. **4**(11), 47–56 (2013)
16. Q Tang, in *ACISP*. LNCS. Towards public key encryption schemes supporting equality tests with fine-grained authorization, vol. 6812 (Springer, Berlin Heidelberg, 2011), pp. 389–406
17. S Goldwasser, Y Kalai, RA Popa, V Vaikuntanathan, N Zeldovich, in *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*. STOC '13. Reusable garbled circuits and succinct functional encryption (ACM, New York, NY, USA, 2013), pp. 555–564. doi:10.1145/2488608.2488678
18. M Bellare, A Boldyreva, A O'Neill, in *CRYPTO*. LNCS. Deterministic and efficiently searchable encryption, vol. 4622 (Springer, Berlin Heidelberg, 2007), pp. 535–552
19. Imagination: MIPS32 Architecture. http://www.imgtec.com/mips/architectures/mips32.asp. last accessed: 07 December 2015 (2015)
20. O Goldreich, *Foundations of Cryptography 1, 2*. (Cambridge University Press, Cambridge/UK, 2003)
21. O Goldreich, R Ostrovsky, Software protection and simulation on oblivious RAMs. J. ACM. **43**(3), 431–473 (1996)
22. B Pinkas, T Reinman, in *CRYPTO*. LNCS. Oblivious RAM revisited, vol. 6223 (Springer, Berlin Heidelberg, 2010), pp. 502–519
23. E Stefanov, E Shi, D Song, in *NDSS*. Towards practical oblivious RAM (arxiv.org, Cornell University, 2012)
24. E Stefanovy, M van Dijk, E Shi, C Fletcher, L Ren, X Yu, S Devadas, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. Path O-RAM: an extremely simple oblivious RAM protocol (ACM, Berlin, Germany, 2013), pp. 299–310. doi:10.1145/2508859.2516660
25. M Maas, E Love, E Stefanov, M Tiwari, E Shi, K Asanovic, J Kubiatowicz, D Song, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. PHANTOM: practical oblivious computation in a secure processor (ACM, New York, NY, USA, 2013), pp. 311–324. doi:10.1145/2508859.2516692
26. B Chor, O Goldreich, E Kushilevitz, M Sudan, in *FOCS*. Private information retrieval (IEEE, 1995), pp. 41–50
27. W Gasarch, A survey on private information retrieval. Bull. EATCS. **82**, 72–107 (2004)
28. C Gentry, Z Ramzan, in *ICALP*. LNCS. Single-database private information retrieval with constant communication rate, vol. 3580 (Springer, Berlin Heidelberg, 2005), pp. 803–815
29. R Ostrovsky, WEI Skeith, in *Public Key Cryptography (PKC)*. LNCS. A survey of single-database private information retrieval: techniques and applications, vol. 4450 (Springer, Berlin Heidelberg, 2007), pp. 393–411
30. D Stehlé, R Steinfeld, in *ASIACRYPT*. LNCS. Faster fully homomorphic encryption, vol. 6477 (Springer, Berlin Heidelberg, 2010), pp. 377–394
31. O Goldreich, *Foundations of cryptography 2: basic applications*. (Cambridge University Press, Cambridge/UK, 2003)
32. S Carpov, P Dubrulle, R Sirdey, Armadillo: a compilation chain for privacy preserving applications. Cryptology ePrint Archive, Report 2014/988 (2014). http://eprint.iacr.org/. Accessed 07 Dec 2015
33. C Aguilar-Melchor, S Fau, C Fontaine, G Gogniat, R Sirdey, Recent advances in homomorphic encryption: A possible future for signal processing in the encrypted domain. Signal Process. Mag. IEEE. **30**(2), 108–117 (2013). doi:10.1109/MSP.2012.2230219
34. NG Tsoutsos, M Maniatakos, in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. Heroic: homomorphically

encrypted one instruction computer (IEEE, 2014), pp. 1–6. doi:10.7873/DATE2014.259
35. W Melicher, S Zahur, D Evans, in *Poster at IEEE Symposium on Security and Privacy*. An intermediate language for garbled circuits, (San Francisco, 2012)
36. S Rass, P Schartner, M Wamser, Oblivious lookup tables. (accepted at the 15th Central European Conference on Cryptology (CECC), available at http://arxiv.org/abs/1505.00605, accessed 07 December 2015 (2015)
37. R Dutra, B Mehne, J Patel, BlindTM—a Turing machine system for secure function evaluation. https://www.cs.berkeley.edu/~kubitron/courses/cs262a-F14/projects/reports/project9_report_ver2.pdf. last accessed: 07 December 2015 (2015)