Towards Using Homomorphic Encryption for Cryptographic Access Control in Outsourced Data Processing

Stefan Rass, Peter Schartner
Universität Klagenfurt, Department of Applied Informatics
email: {stefan.rass, peter.schartner}@aau.at

***Abstract*—We report on a computational model for data processing in privacy. As a core design goal here, we will focus on how the data owner can authorize another party to process data on his behalf. In that scenario, the algorithm or software for the processing can even be provided by a third party. The goal is here to protect the intellectual property rights of all three players (data owner, execution environment and software vendor), while retaining an efficient system that allows data processing in distrusted environments, such as clouds. We first sketch a simple method for private function evaluation. On this basis, we describe how code and data can be bound together, to implement an intrinsic access control, so that the user remains the exclusive owner of the data, and a software vendor can prevent any use of code unless it is licensed. Since there is no access control logic, we gain a particularly strong protection against code manipulations (such as “cracking” of software).**

Keywords*—*private function evaluation; cloud computing; licensing; security; cryptography.

I. Introduction

Cloud computing is an evolving technology, offering new services like external storage and scalable data processing power. Up to now, most cases of data processing, such as statistical computations on medical data, are subject to most stringent privacy requirements, making it impossible to have third parties process such person-related information.

A classical technique to prevent unauthorized parties from reading confidential information is by use of encryption. Unfortunately, this essentially also prevents any form of processing. This work concerns a generic extension [1] to standard ElGamal encryption, towards enabling permitted parties to process encrypted information without ever gaining access to the underlying data.

The core of this paper is a mechanism to endow the data and software owner with the capability of allowing or preventing designated parties from using either the data or the software for any data processing application. This is to let users retain full control over their data and software. The licensing scheme described herein is thus a method of providing or revoking the explicit consent to data processing in privacy. Moreover, unlike classical access control techniques, our scheme is cryptographic and as such cannot be circumvented nor deactivated by standard hacking techniques.

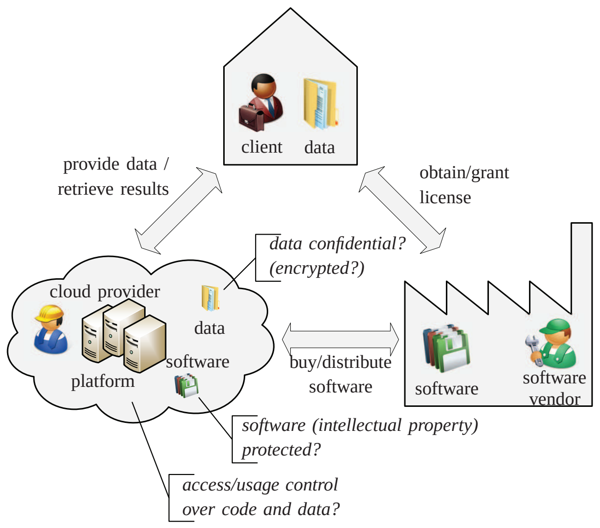

Figure 1. Example Scenario – Cloud Computing.

The most general scenario to which our licensing scheme (and computing model) applies involves three entities: first, there is the *client* (CL), who owns data that needs processing. The second player is the *software vendor* (SV), who owns the code for data processing. The third party is the *execution environment* (EE), which is the place where the actual data processing takes place (e.g., a cloud provider with sufficient hardware resources, or similar).

Figure 1 illustrates an example scenario, in which a client hands over its data to a cloud provider who runs third-party software for data processing services. Security issues are printed in italics.

Especially the client and software vendor have different interests, which may include (but are not limited to) the following:

- The client wants to keep its data confidential and wants to keep control over how and where it is processed
- The software vendor wants to prevent theft of its computer programs (software piracy), or other misuse of its software by unauthorized parties

The execution environment can be seen as the *attacker* in our setting: it is the only party that has access to both, the data and the algorithms to process it. So, its main interest would be gaining access to the data, or run the program on data of its own supply. We emphasize that the described protection does not automatically extend to the algorithm itself. However, it is a simple yet unexplored possibility to apply code obfuscation in the computational model that we sketch in Section I-A.

Based on the above division, we can distinguish the following four scenarios:

1) All three entities separated: in this setting, the EE runs an externally provided software from the SV on data provided by the CL.
2) SV = EE: an example instantiation of this setting would be cloud SaaS, such as GoogleDocs. Here, the client obtains a licence to use a particular software, but seeks to protect his data from the eyes of the (cloud) provider.
3) CL = SV: here, the client is the one to provide the code for data analysis, yet seeks to outsource the (perhaps costly) computation to an external entity, e.g., a cloud provider.
4) CL = EE: the client obtains the software from the SV and runs the code on its own data within its own premises. Here, actually no particular licensing beyond standard measures is required (not even encrypted code execution), so we leave this scenario out of further investigations.

A. The Basic Idea – Outline of the Main Contributions

Briefly sketching what comes up, we will describe how algorithms can be executed on encrypted data, using a a *blind Turing machine* (BTM) [1]. Leaving the details of BTMs aside here (for space reasons), the central insight upon which this work is based is the fact that BTMs require a secret encoding of the data, which establishes compatibility between the data and the program that processes it. More specifically, BTMs, in the way used in this paper, allow the execution of arbitrary assembly instructions on encrypted data. Briefly (yet incompletely) summarizing the idea posed in [1], we encrypt a data item x into a pair $(E_{pk_1}(x), E_{pk_2}(g^x))$, where pk_1, pk_2 are two distinct public keys, g^x is a cryptographic commitment to x, and E is any public key encryption. The crux of this construction is the possibility of comparing two encrypted values $x_1 \overset{?}{=} x_2$, without revealing either value, based only on decryptions of the commitments $x_1 = x_2 \iff g^{x_1} = g^{x_2}$. Herein, neither commitment reveals x_1 or x_2, if computing discrete logarithm computations are intractable in the underlying group of E (the trick is similar yet with a different goal as for commitment consistent encryption; cf. [2]). Hereafter, we will use a subgroup of prime order q within the set $\mathbb{Z}_p$, when p is a large safe prime.

Executing arithmetic assembly instructions like `add A, B, C`, where $A \leftarrow B+C$ and B, C are encrypted values, computing the sum (or any other operation like multiplications, logical connectives, etc.) can be done by a humble table-lookup, based on the equality checking of encrypted inputs. Equally obvious is that the necessary lookup tables have to be small, i.e., we have only a small number of inputs $\{x_1, \ldots, x_n\}$. Practically, n is limited to small values of n, to keep the lookup tables (of size $O(n^2)$ feasibly small). Indeed, this is still an advantage of many fully homomorphic encryption schemes, which work on the bit-level (where we would have $n = 2$ for $x_1 = 0$ and $x_2 = 1$ in our setting).

The smallness of the plaintext space, together with the equality checking of the (so-modified) encryption scheme, also enables attacks by brute-force trial encryptions (of $x_1, \ldots, x_n$) and equality checks of the candidate plaintext to decipher any register content. Thwarting this attack is simple, if the encryption additionally uses a secret random representative a to encode the input before encrypting it (thus taking away the adversary's ability to brute-force try all possible plaintexts). That is, the encryption of x is actually one of $a \cdot x$. To ease notation in the following, we write $E_{pk}(g^{a \cdot x})$ as a shorthand of the secret message x being encoded with the random value a, where the encoding is

$$x \mapsto g^{ax}. \tag{1}$$

As a technical condition, we require $\gcd(a, p-1) = 1$.

Our description of the computational model is admittedly somewhat incomplete, as we do not discuss how memory access or control flow can be handled in the blind Turing machine model (when applied to assembly instruction executions). We leave this route for further exploration along follow up research, and confine ourselves to the observation that code (involving encrypted constants like offsets for memory access, etc.) and data can be made compatible or incompatible, based on whether the secret encoding used for the code (a value a) and the data (another value b) is equal or not.

The rest of the paper will be devoted to changing the secret value a – the *encoding* – or negotiating it between two or three parties (CL, SV, EE). The respective protocols form the announced *licensing* scheme, which are nothing else than the *authorization* to use the encryption's plaintext comparison facility. Practically, knowledge of a and the comparison keys (the secret decryption key sk_2 belonging to pk_2, to decrypt the commitments) enable (or in absence disable) the ability to run an arbitrary algorithm on encrypted data.

The authorization is thus bound to knowledge of an *evaluation key*, which is composed from the comparison token (secret key sk_2), plus the lookup tables (for all assembly instructions). The encoding a is *excluded* from the evaluation key, so that it can be given to the EE without enabling it to process data of its own interest.

Blind Turing machines provide a technical possibility to do the following upon a combination with the licensing scheme as described in Section III:

1) Encrypt a software in a way so that only licensed copies of it can be run on input data. This is *security for the software vendor*, in the sense of preventing software use without license, e.g., by the EE.
2) Encrypt data in a way to bind its use to a single licensed copy of a software (so that data processing by unauthorized parties is cryptographically prevented). This

is *security for the client*, in the sense of preventing misuse of either her/his software license or her/his data as given to the EE.

B. Example Applications

We briefly describe three possible applications, leaving more of this for extended versions of this work.

Cloud Services for Data Processing: consider an online service that offers data processing over a web-interface, using a software that runs remotely within the cloud. The client could safely input its data through the web-interface, knowing that the cloud is unable to execute the program (for which the client has obtained a license) on other data that what comes from the client. In addition, the client can be sure that the cloud provider does not learn any of the secret information that the customer submits for processing.

Such services are already existing, although not at the level of security that we propose here. One example is *Google Docs*.

En-route information processing: sensor networks and smart metering infrastructures use decentralized data processing facilities. In case of *smart metering*, data concentrators collect and preprocess data harvested from the subscribers, before sending properly compiled information to the head end for further processing (such as billing, etc.). Using the proposed licensing scheme, this processing could be done in entirely encrypted fashion, without "opening or breaking" the encrypted channel for the sake of intermediate processing.

A third example scenario is the **protection of intellectual property**, namely the firmware that runs inside a device. This example is expanded in full detail in Section V.

C. Organization of the Paper

Section II discusses related work. Section III is based on the model for private function evaluation as sketched in Section I-A, and describes the ideas underneath the main contribution as described in Section IV. That section also completes the description of how the authorization is implemented and granted. Security of our protocols is discussed in Section VI, and concluding remarks are made in Section VII.

II. Related Work – Computation in Privacy

Processing encrypted data is traditionally done using one of three approaches: homomorphic encryption, multiparty computation (MPC) and garbled circuits (GC). Picking up homomorphic encryption as the most recent achievement, many well-known encryption schemes are homomorphisms between the plain- and ciphertext spaces. Prominent examples are RSA and ElGamal encryption, which are both multiplicatively homomorphic. Likewise, Paillier encryption [3] enjoys an additive homomorphic property on $\mathbb{Z}_n$, where n is a composite integer (as for RSA), and the Goldwasser-Micali encryption [4], which is homomorphic w.r.t. the bitwise XOR-operation.

Surprisingly, until 2009 no encryption being homomorphic w.r.t. to more than one arithmetic operation was known. The work of Gentry [5] made a breakthrough by giving an encryption that is homomorphic for both, addition and multiplication. Ever since this first *fully homomorphic encryption* (FHE), many variations and improvements have appeared (e.g., [6]–[8] to name a few), among these being *somewhat homomorphic encryptions*, which permit several arithmetic operations, however, only a limited number of executions of each operation (e.g., arbitrarily many multiplications, but only a one addition over time).

Yao's concept of *garbled circuits* [9] provides a way to construct arithmetic circuits that hide their inner information flow by means of encryption. Interestingly, this works without exploiting any homomorphism, and is essentially doable with most standard off-the-shelf encryption primitives (cf. [10] and references therein).

Common to these two mainline approaches to the problem of data processing in confidentiality is the need to construct evaluation circuits (for both, fully homomorphic encryption and garbled circuits) that strongly depend on the data processing algorithm. In that sense, neither technique offers a fully automated mechanism to put an arbitrary algorithm to work on encrypted information, and compilers that take over this task are subject of intensive ongoing research [11]–[17]. Similar difficulties apply to multiparty computation approaches [18]–[20] or combinations of GC and MPC [10].

In the past, encryption circuits or interactive protocols have commonly been used as computational models, as opposed to Turing machines, which have only recently been considered as an execution vehicle [1], [21]. The latter of these references proposed the concept of a *blind Turing machine*, which is an entirely generic construction that uses standard ElGamal encryption (unlike [21], which works with attribute-based encryption). The idea relies on Turing machines as the most powerful known computational model (up to other models being equivalent to Turing machines), and the construction resembles the full functionality of a general Turing machine using encrypted content. This approach has briefly been outlined in Section I-A.

III. The Licensing Scheme

The main objective of the licensing protocols is to change the value a that encodes the secret data item $x \in \mathbb{Z}_p$ by (1). Hereafter, we let $\in_R$ denote a uniformly random draw from the given set. From the construction of blind Turing machines, i.e., the execution of instructions by table-lookups on the data being processed, it is evident that a program can only be executed if the code and data obey the same encoding (since the lookup table in the evaluation key must use the same encoding a as the data, for otherwise the lookup will fail). Establishing or changing the common encoding a is detailed in the next subsection.

A. Changing the Encoding

If a is known, then, it is easy to switch to another encoding based on b, via raising (1) to the power of $a^{-1}b$, where a^{-1} is computed modulo $p-1$ (this inverse exists, as we assumed a relatively prime to $p-1$; see Section III). This gives

$$(g^{ax})^{a^{-1}b} \equiv g^{axa^{-1}b} \equiv g^{bx} \pmod{p}. \tag{2}$$

B. Negotiating an Encoding

If a given encoding of one entity (e.g., the SV) shall be changed to a chosen encoding of another entity (e.g., the CL), then the following interactive scheme can be used to switch from encoding a to encoding b, while revealing neither value to the other party. The protocol is as follows, where entity A secretly knows the encoding a, which shall be changed into the encoding b that entity B secretly chose. Common knowledge of both parties are all system parameters, in particular the generator g and prime p are known to both parties A and B.

1) $A \to B$: an encoded item g^{ax}.
2) $B \to A$: raise g^{ax} to b, and return the value $(g^{ax})^b \equiv g^{abx} \pmod{p}$.
3) A: strip a from the exponent via $(g^{axb})^{a^{-1}} \equiv g^{bx} \pmod{p}$. A can continue to work with the new encoding b, which is in turn unknown to A.

Notice that the knowledge of A is x, a, g^x, g^{ax} and g^{abx}, from which b cannot be extracted efficiently.

IV. Putting it to Work

With the encoding taking the form (1) and the encryption E being *multiplicatively homomorphic* (e.g., ElGamal), we can apply Diffie-Hellman like protocols to change the values in the exponent $g^{a\cdot x}$ even within an encryption. In the following, it is important to stress that any communication between the entities in the upcoming scenarios is encrypted, in order to prevent external eavesdroppers from trivial disclosure of secret information (an evident possibility in the protocols).

1) Licensing Scenario 1: Three Separated Parties: Suppose that a program P written by the SV resides within the EE under an encoding a (unknown to the EE). We assume that the program P is encrypted under the SV's public key pk_{SV} for reasons of intellectual property protection and to effectively prevent an execution without explicit permission by the SV.

To obtain a license (permission) and execute the program, the following steps are taken (Figure 2 illustrates the process in alignment to Figure 1).

1) The CL initiates the protocol by asking SV for a license.
2) The SV chooses a secret value $b \in \mathbb{Z}_p$ and sends the quantity $a^{-1}b \mod (p-1)$ to the EE, which it can use to "personalize" the program P by re-encoding it as

$$E_{pk_{SV}}(g^{ax})^{a^{-1}b} = E_{pk_{SV}}((g^{ax})^{a^{-1}b}) = E_{pk_{SV}}(g^{bx}), \tag{3}$$

by virtue of the multiplicative homomorphy of $E_{pk_{SV}}$.
3) The CL prepares the evaluation key, i.e., the respective lookup-tables under *his own* public key pk_{CL}. This implies that all results obtained from the lookup table can only be decrypted by CL after the computation has finished. In particular, it assures that all internal intermediate results obtained over the execution of P will be encrypted under the CL's public key, so that they remain inaccessible for the EE or the SV.
4) Using b the CL can encode and submit its data to the EE for processing. The results are encrypted under pk_{CL} and hence only accessible to the CL afterwards.

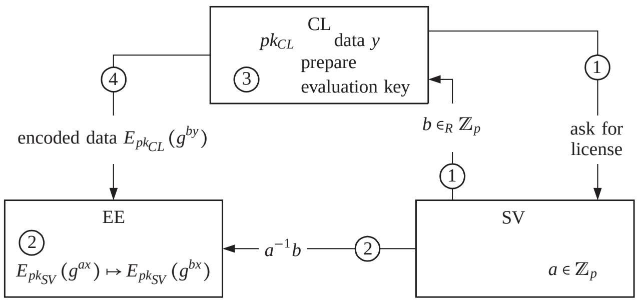

Figure 2. Licensing scenario involving three separated parties.

It is obvious that the scheme becomes insecure if two out of three of these entities collaborate in a hostile fashion. In either case, the secret encoding and also the secret data could be disclosed.

2) Licensing Scenario 2: SV = EE: Here, the EE/SV knows the encoding a but the client can interactively change it into his own chosen encoding b to obtain a license. Referring to Section III-B for the details, the remaining steps comprise the execution of the program P, which is then compatible with the secret encoding b under which the data has been prepared. For personalization, the SV/EE decrypts and submits all code items $E_{pk_{SV}}(g^{ax})$ to the client for re-encoding. Note that the CL *cannot* run the program, as it lacks the code itself (the CL gets only the constants found in the code). Figure 3 illustrates the details.

3) Licensing Scenario 3: CL = SV: This case is even more trivial, as the CL, being the SV at the same time, simply chooses the encoding a and submits its code and data to the EE for processing. No change or interactive negotiation of encoding is required here.

4) Involving a Different End-User: In some cases, the CL may be the source but not the final end-user of the data (e.g., in a smart meter network, where the CL is a user's smart meter, the EE is a data aggregator/data concentrator, and the end-user is the energy provider's head end system). In such cases, it is straightforward to prepare the evaluation key for a (fourth) party EU (*end user*). The change is simply by preparing the evaluation key under the EU's public key pk_{EU} instead of pk_{CL}. With this modification, all of the above scenarios work exactly as described.

V. Example Application Scenario

Another potentially very important application of our licensing scheme concerns the *protection of firmware (intellectual property)*. Suppose a manufacturer – here being the client CL – obtains the device's firmware from an external software vendor

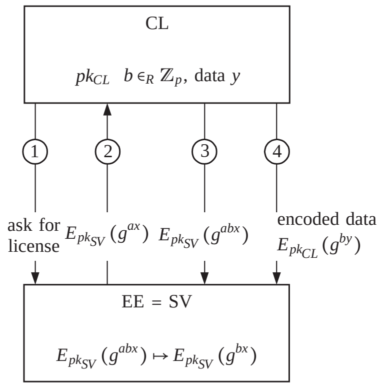
CL
pk_{CL} $b \in_R \mathbb{Z}_p$, data y
1
2
3
4
ask for license
$E_{pk_{SV}}(g^{ax})$
$E_{pk_{SV}}(g^{abx})$
encoded data $E_{pk_{CL}}(g^{by})$
EE = SV
$E_{pk_{SV}}(g^{abx}) \mapsto E_{pk_{SV}}(g^{bx})$

Figure 3. Licensing scenario when the data processing and software remains with the SV.

(SV). Furthermore, assume the device is equipped with an internal unique identity, such as a physically uncloneable function (PUF) or other hardwired unchangeable and uncloneable identity. We call this identity *ID*.

As before, let the firmware be code with encrypted fragments of the form $E_{pk_{SV}}(g^{ax})$, under a secret encoding $a \in \mathbb{Z}_p$ used by the SV, which is unknown to CL. After uploading the firmware, the device manufacturer obtains a license by

1) choosing a secret value β and submitting the blinded identity $\beta \cdot ID \text{ MOD } p$ to SV, and
2) retrieving the license $L' = (g^{b \cdot ID \cdot \beta}, a^{-1} \cdot b \cdot (ID^2) \cdot \beta^2 \text{MOD } p)$ from the firmware manufacturer, where $b \in \mathbb{Z}_p$ is a secret random value chosen by the SV. The device manufacturer then strips the factors β, resp. β^2, from the contents of L' to obtain the final license $L = (\ell_1, \ell_2) = (g^{b \cdot ID}, a^{-1} \cdot b \cdot (ID^2) \text{MOD } p)$.

We note that blinding β is only required to avoid attackers listening on the channel in an attempt to clone an identity and hence a device, runnable with the same license as for the honest manufacturer. Using β, the license L cannot be obtained from L' unless by the device manufacturer (or the device itself), knowing β.

Using the license L, the CL can personalize the code via

$$\left(E_{pk_{SV}}(g^{ax})\right)^{\ell_2} = \left(E_{pk_{SV}}(g^{ax})\right)^{a^{-1}bID^2} = E_{pk_{SV}}(g^{b \cdot ID^2 \cdot x}), \quad (4)$$

and can encode input data y accordingly by computing

$$\ell_1^{y \cdot ID} \equiv (g^{b \cdot ID})^{y \cdot ID} \equiv g^{b \cdot ID^2 \cdot y} \pmod{p}. \quad (5)$$

We stress that an extracted firmware (e.g., software piracy) *will not* run on a structurally identical hardware, as the other device works with a different $ID' \neq ID$, even if the same license $L = (\ell_1, \ell_2)$ is brought into the device!

To see this, observe that the encoding is actually $b \cdot ID^2$, yet the encoding information for the data is only $g^{b \cdot ID}$, which enforces an exponentiation with the internally supplied identity (e.g., PUF-value) *ID*. Hence, the encoding by (5) will fail to reproduce ID^2 in the exponent, as *ID* cannot be replaced in the second device (as coming from a PUF for example). The exponent, in that case, will take the form $g^{b \cdot ID \cdot ID' \cdot y}$, which is incompatible with the program encoded via $g^{b \cdot ID^2}$.

VI. Security

By construction and the discussion in Section I-A, our scheme becomes insecure under any of the following two circumstances:

1) collaboration of at least two entities in any of the described licensing scenarios
2) a party succeeds in extracting the constant a that defines the secret encoding of symbols as defined in (1).

Obviously, we cannot mathematically rule out hostile cooperations among any of the entities in our context, but we can prove that the second of the above attack scenarios will fail under usual computational intractability hypotheses.

More concretely, we prove security of our licensing scheme by showing that the extraction of a license is at least as hard as computing discrete logarithms in the underlying group (see Definition VI.1). To this end, we distinguish different potential attackers and licensing scenarios according to our preceding discussion. Throughout this section, we assume passive adversaries and authenticated parties (thus, we do not discuss person-in-the-middle scenarios here).

Definition VI.1 (Discrete Logarithm Problem)**. Given:** *a prime p, a generator g of $\mathbb{Z}_p^*$ and a value $y \in \mathbb{Z}_p^*$.*
Sought: *an integer $x \in \mathbb{N}$, such that $y = g^x \, MOD \, p$. We write $x = dlog_g(y)$ and call this the* base-*g*-logarithm *of y.*

We call this problem intractable*, if there is no efficient algorithm able to compute the base-g-logarithm of y.*

Under the intractability of discrete logarithms, security of our scheme is easy to prove in every scenario. Observe that the encryption wrapped around the values considered in the following can be neglected in cases where the attacker is an "insider", i.e., the client CL, the execution environment EE, or an external person-in-the-middle intruder.

A. Licensing Scenario 1: Three Separated Parties

Precluding collaborations between parties, the attacker can either be the client, the execution environment or an external eavesdropper. Consequently, we need to analyze security in each case separately.

This case is essentially trivial, as the client CL gets only his personal license b, but cannot access the encrypted quantity $a^{-1}b$ that is sent directly to the execution environment. As b is chosen stochastically independent of a, it does not provide any information about a.

Under a slight modification by sending g^b instead of b in this scenario, we can even allow an attacker to mount a person-in-the-middle attack, in the course of which he gets $a^{-1}b$ and g^b in plain text. The following result asserts security even under this modified stronger setting.

Proposition VI.2 (Security against external adversaries)**.** *Let $a \in \mathbb{Z}_p$ be the secret license used by the software vendor, and let $I = (a^{-1}b, g^b)$ be the attacker's information. Computing a from I is at least as hard as computing discrete logarithms.*

Proof. Let A be an algorithm that extracts a from $I = (a^{-1}b, g^b)$. We construct another algorithm A' that computes discrete logarithms and takes only negligibly more time for this than A needs. Given a value y, algorithm A' simply submits the pair (z, y) to A, where z is a uniformly random number. As b is uniquely determined by $y = g^x$, there is another unique number z' that satisfies $z = z' \cdot x$, where z' is stochastically independent of x. Hence, the pair (z, y) has the proper distribution to act as input to A, and A returns z' so that the sought discrete logarithm of y returned by A' is $x = z' \cdot z$. □

By symmetry, security against a malicious execution environment EE holds by the same line of arguments, and under both, the modified and original licensing scenario. The problem for a malicious EE is to compute the client's license b from its information $I = (a^{-1}b, g^{by})$, which is even harder as before, as there is another stochastically independent quantity y that blinds b in that case. We hence get the following result, whose proof is obvious from the preceding discussion:

Proposition VI.3 (Security against malicious EE)**.** *Let $b \in \mathbb{Z}_p$ be the secret license of the CL, and let $I = (a^{-1}b, g^{yb})$ be the attacker's information. Computing b from I is at least as hard as computing discrete logarithms.*

B. Licensing Scenario 2: SV = EE

Here, the problem is to extract b from $(g^{ax}, g^{abx}, g^{by})$. Given that the software vendor is identical to the execution environment, we can assume the attacker to know the values x and a, so that the actual problem is to compute b from $I = (g^b, g^{by})$.

Proposition VI.4 (Security in case of malicious SV=EE)**.** *Let p, q be primes so that $p = 2q + 1$ and let g generate a q-order subgroup of $\mathbb{Z}_p$. Computing the client's secret b from $I = (g^b, g^{by})$ is at least as hard as computing discrete logarithms in the subgroup $\langle g \rangle \subsetneq \mathbb{Z}_p$.*

Proof. The argument is again a reduction: let A be an algorithm that correctly returns b upon input $I = (g^b, g^{by})$. We construct an algorithm A' that computes discrete logarithms as follows: given a value y, we submit the input (y, z) to A, where z is a random value. As $y = g^x$ uniquely defines a value x, it also uniquely defines a value z' so that $z = g^{xz'}$. To see this, observe that the solvability of the equation $g^{by} = z$, by taking discrete logarithms on both sides, is equivalent to the solvability of congruence $by \equiv \mathrm{dlog}_g z \pmod{q}$, which is trivial. Hence, (y, z) has the proper input-distribution for A, which then correctly returns the discrete logarithm x of $y = g^x$. □

We stress that working in subgroups is not explicitly assumed in the previous security proofs, yet is a standard requirement in secure instantiations of ElGamal encryptions, such as over elliptic curves. Hence, the additional hypothesis of proposition VI.4 is mild and will always be satisfied in practical scenarios.

C. Licensing Scenario 3: CL = SV

Here, the problem is for the EE to extract information from the data submitted by the client. This is equivalent to either breaking the cipher or computing discrete logarithms, and hence covered by known security proofs concerning the underlying cryptographic concepts.

VII. Conclusion and Outlook

This work is compilation of concepts that enable secure and authorized processing of encrypted information. In essence, it is a deployment scheme for private function evaluation based on blind Turing machines, where the involved parties can secure their interests (prevention of software piracy and prevention of personal data misuse) by running interactive protocols.

Along experiments with implementations of the ideas sketched here, we identified various security issues and possible attacks, some of which were sketched in the previous sections. Future work is on implementing the described ideas and studies of security implications on a real prototype implementing a full computing platform. To this end, the concept of oblivious lookup tables [22] has been devised as a substitute that does lookups without comparing encrypted plaintexts.

Unfortunately, chosen instruction attacks are not entirely disabled in that case, since branching instructions and memory access can be turned into a plaintext comparison facility (e.g., by submitting conditional branches and observe the control flow, or by asking for two ciphertexts to address the same memory cell, using the fact that the physical addressing is most likely deterministic). A working prevention against such misuse calls for additional code obfuscation (particularly on the branch instructions) and secure memory access techniques (such as oblivious RAM or private information retrieval). Interestingly, this renders the proof of security of blind Turing machines against active adversaries (see [1]) practically void, as the assumptions of the proof are violated in side-channel scenarios.

As an overall conclusion, however, the following points can be made:

- Processing encrypted information *appears possible* using standard encryption, although this induces new vulnerabilities like side-channel information leakage. Whether ultimate security can be achieved in this generic construction (as incompletely sketched in Section I-A) is an interesting open issue; we hope that this article stipulates future research in this direction.
- Authorized processing of data can be achieved in various settings by agreeing on secret encodings and/or changing them interactively by exploiting the homomorphy of encryption, as described in Section III.

- The efficiency of our licensing scheme depends on how large the code is that we "personalize", since every instruction of a program and every data item has to be (re-)encoded. The main practical bottleneck will, however, be the underlying data processing system; in case of blind Turing machines, this amounts to roughly 1 multiplication and 1 exponentiation per instruction (processing up to 4 bits). See [23] for a detailed analysis and comparison to competing approaches.

It is important to note that any of the described schemes can be implemented in general groups; there is no need to strictly rely on modulo-arithmetic (this particular instantiation serves only illustrative purposes). Hence, for a practical implementation, we recommend elliptic curve groups (elliptic curve cryptography) or similar as a substitute for the structure $\mathbb{Z}_p$.

Most importantly, the scheme works mostly using off-the-shelf cryptographic primitives that have been known for decades, are well understood and enjoy good hardware support already.

References

[1] S. Rass, "Blind turing-machines: Arbitrary private computations from group homomorphic encryption," International J. of Advanced Computer Science and Applications, vol. 4, no. 11, pp. 47–56, 2013.

[2] E. Cuvelier, O. Pereira, and T. Peters, "Election verifiability or ballot privacy: Do we need to choose?" Cryptology ePrint Archive, Report 2013/216, 2013, http://eprint.iacr.org/.

[3] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in Proc. of EUROCRYPT'99, ser. LNCS, vol. 1592. Springer, 1999, pp. 223–238.

[4] S. Goldwasser and S. Micali, "Probabilistic encryption," Special issue of J. of Computer and Systems Sciences, vol. 28, no. 2, pp. 270–299, April 1984.

[5] C. Gentry, "Fully homomorphic encryption using ideal lattices," in Proc. of STOC'09. New York, NY, USA: ACM, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[6] C. A. Melchor, P. Gaborit, and J. Herranz, "Additively Homomorphic Encryption with *d*-Operand Multiplications," in CRYPTO, ser. LNCS, vol. 6223. Springer, 2010, pp. 138–154.

[7] A. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters, "Fully secure functional encryption: attribute-based encryption and (hierarchical) inner product encryption," in Prof. of EUROCRYPT'10. Berlin, Heidelberg: Springer, 2010, pp. 62–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13190-5_4

[8] D. Boneh, E. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in Proc. of TCC'05, LNCS 3378, 2005, pp. 325–341.

[9] A. C.-C. Yao, "How to Generate and Exchange Secrets (Extended Abstract)," in FOCS. IEEE, 1986, pp. 162–167.

[10] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in 20th USENIX Security Symp. USENIX Assoc., 2011.

[11] N. Tsoutsos and M. Maniatakos, "HEROIC: homomorphically encrypted one instruction computer," in Proc. of (IEEE DATE'14), March 2014, pp. 1–6.

[12] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: a compilation chain for privacy preserving applications," Cryptology ePrint Archive, Report 2014/988, 2014, http://eprint.iacr.org/.

[13] C. Aguilar-Melchor, S. Fau, C. Fontaine, G. Gogniat, and R. Sirdey, "Recent advances in homomorphic encryption: A possible future for signal processing in the encrypted domain," Signal Processing Magazine, IEEE, vol. 30, no. 2, pp. 108–117, March 2013.

[14] M. Brenner, J. Wiebelitz, G. von Voigt, and M. Smith, "Secret program execution in the cloud applying homomorphic encryption," in Proc. of IEEE DEST, May 2011, pp. 114–119.

[15] V. Kolesnikov and T. Schneider, "A practical universal circuit construction and secure evaluation of private functions," in Prof. of FC, G. Tsudik, Ed. Springer, 2008, pp. 83–97. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85230-8_7

[16] W. Melicher, S. Zahur, and D. Evans, "An intermediate language for garbled circuits," in Poster at IEEE Symp. on Security and Privacy, 2012.

[17] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in Proc. of STOC'13. New York, NY, USA: ACM, 2013, pp. 555–564. [Online]. Available: http://doi.acm.org/10.1145/2488608.2488678

[18] Z. Beerliová-Trubíniová and M. Hirt, "Perfectly-secure MPC with linear communication complexity," in Proc. of TCC'08. Berlin, Heidelberg: Springer, 2008, pp. 213–230. [Online]. Available: http://dl.acm.org/citation.cfm?id=1802614.1802632

[19] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: Tool for Automating Secure Two-Party Computations," in CCS. ACM, 2010, pp. 451–462.

[20] Y. Lindell, B. Pinkas, and N. P. Smart, "Implementing two-party computation efficiently with security against malicious adversaries," in Proc. of SCN'08. Springer, 2008, pp. 2–20. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85855-3_2

[21] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "How to run turing machines on encrypted data," Cryptology ePrint Archive, Report 2013/229, 2013, http://eprint.iacr.org/.

[22] S. Rass, P. Schartner, and M. Wamser, "Oblivious lookup tables," 2015, accepted at the 15th Central European Conference on Cryptology (CECC), http://arxiv.org/abs/1505.00605.

[23] S. Rass, P. Schartner, and M. Brodbeck, "Private function evaluation by local two-party computation," EURASIP Journal on Information Security, vol. 2015, no. 1, 2015. [Online]. Available: http://dx.doi.org/10.1186/s13635-015-0025-9