Secret Program Execution in the Cloud Applying Homomorphic Encryption

Michael Brenner, Jan Wiebelitz, Gabriele von Voigt and Matthew Smith
Research Center L3S, Hannover, Germany
{brenner,wiebelitz,vonvoigt,smith}@l3s.de

Abstract—A growing number of compute and data storage jobs is performed on remote resources. In a cloud environment the customer can’t be sure where a particular job is physically executed and thus cannot rely on the security and confidentiality of the remote resource. A solution for this problem is operating on encrypted functions and encrypted data. This enables a customer to generate a program that can be executed by a third party, without revealing the underlying algorithm or the processed data. This helps securing applications and data in a distributed digital ecosystem.

We present a method to compute a secret program on an untrusted resource using fully homomorphic encrypted circuits. We sketch an algebraic homomorphism as a cryptographic foundation and define a sample system architecture for which we provide a software implementation. Our concept solves the problems of encrypted storage access with encrypted addresses and encrypted branching: in contrast to other approaches, like static one-pass circuit simulations, our system supports dynamic parameters and non-linear programs, that render branch-decisions at runtime and cannot be represented in a circuit with hard-wired in-circuit parameters and data. Our implementation comprises the runtime environment for an encrypted program and an assembler to generate the encrypted machine code.

Index Terms—homomorphic encryption, secure function evaluation, encrypted program execution, encrypted memory access, encrypted branching, mobile code security

I. INTRODUCTION

The ability to securely delegate computation to a remote resource provider is becoming a key feature in resource outsourcing and cloud computing, where programs and data are distributed over a wide network of machines and resources that can no longer be controlled by the customer. The customer has to trust the resource provider, because to date all software - once transmitted to the remote resource provider - is executed in unencrypted form and is entirely under the control of the resource owner. There is a need for a mechanism to operate on encrypted programs and data.

A solution to this is fully homomorphic encryption, which has often been called the cryptographer’s holy grail. Once the mathematical foundation has been established [1], we need further procedures and an architecture that enable a reasonable application of encrypted additions and multiplications on single bits. It’s essentially this, what the latest homomorphic systems provide. Our goal is the construction of a generic runtime container that is capable of executing arbitrary encrypted programs, operating on encrypted data.

In this paper we present a method to compute an encrypted program on an untrusted resource using fully homomorphic encrypted circuits. To avoid the reader’s distraction and for a more readable discourse, we introduce a simple algebraic homomorphism as a cryptographic foundation to present our concept. We define a sample system architecture for which we provide a software implementation. Our concept solves the problems of encrypted storage access with encrypted addresses and encrypted branching: in contrast to other approaches, like Yao’s Garbled Circuits [2] and several extensions such as [3] and [4], our system supports non-linear programs, dynamic parameters and subsequent provision of encrypted input data that can easily be written into the encrypted memory. We support programs that render dynamic branch-decisions at runtime, even allow self-modifying code and thus cannot be represented in a one-pass circuit. We achieve this by applying a ciphertext lock up strategy that seals program code and data entirely in the encrypted domain, which is a closed algebraic system. Our concept also solves the problem of both protecting an executing host from malicious code and protecting mobile code from a malicious host.

The paper structure is as follows: Related work and other interesting approaches are discussed in Section II. Section III gives a brief overview of homomorphic encryption and describes a simple algebraic homomorphism that is used as a reference model for the description of our concept. Section IV introduces our approach of homomorphically encrypting circuits using an integer representation. We also introduce processor primitives that are described in detail, both in boolean and encryptable integer logic. Section V gives an overview of different use cases for our system in a distributed or cloud environment. Applying the foundation of encrypted circuits, we discuss our software implementation and provide basic performance figures in Section VI. Our future work and lessons learned from the implementation are presented in Section VII. We give a short summary of our contributions in Section VIII.

II. RELATED WORK

Most existing cryptographic methods aim at securing the transport and storage of data. With the advent of Cloud Computing, new approaches emerged and older ones have been revisited. One group of concepts to secure computation is the construction of boolean circuits as a function representation. The circuit is then encrypted in some way (actually, in cases of Yao derivatives, the function tables of the boolean gates are assigned different values) to conceal the original function.
The contributions of Yao [2], Abadi [5] and Malkhi [3] are examples for this class of approaches. The goal is to establish a protocol and an execution container that allows for Secure Function Evaluation of a common function with a secret input from every participating entity. Another field is Mobile Code Protection (MCP). Two directions can be observed here: A) the protection of code from a malicious host and B) the protection of a host from malicious, mobile code. MCP bases on policies and reduced access rights of execution containers. An implementation of method A for a Java environment is described in [10], a more general method in [11]. Approaches for case B often rely on a defined and controllable platform and aim in the direction of hardware supported Trusted Platform Computing [12]. A working secure system, based on TPC is Turaya [15] which implements secure application compartments to protect the host and to separate applications from each other. A third class of concepts addresses security by evaluating encrypted functions and/or data, where most methods apply some sort of homomorphic encryption. Sander and Tschudin [13] [14] have proposed a scheme that is able to evaluate encrypted polynomials over rings \( \mathbb{Z}/n\mathbb{Z} \). Lee et al have been working on a method to partially encrypt and evaluate a series of interdependent three-address statements [6]. Advanced homomorphic methods apply some sort of homomorphic encryption. Sander \[15\] which implements secure application and Tschudin [13] [14] have proposed a scheme that is able to evaluate encrypted polynomials over rings \( \mathbb{Z}/n\mathbb{Z} \).

### III. Homomorphic Encryption

A homomorphism is a structure-preserving transformation between two sets, where an operation on two members in the first set is preserved in the second set on the corresponding members.

Let \( P \) and \( C \) be sets with members \( p_1, p_2 \in P, t \) a transformation between the two sets with its reverse function \( t' \) and an operation \( \oplus \). The system is a homomorphism, if \( \forall p_1, p_2 \in P, (p_1 \oplus p_2) = t'(t(p_1) \oplus t(p_2)) \). If there are two functions \( \oplus \) and \( \otimes \), such that \( \forall p_1, p_2 \in P, (p_1 \oplus p_2) = t'(t(p_1) \oplus t(p_2)), (p_1 \otimes p_2) = t'(t(p_1) \otimes t(p_2)) \), we call this an algebraic homomorphism [6]. The obvious practical implication is the possibility to transform the two members \( p_1 \) and \( p_2 \) into the range of \( C \), thus applying some sort of encryption, and having the operations \( \oplus \) and \( \otimes \) performed by a third party. The result can then be decrypted back into the range of \( P \).

An algebraically homomorphically crypto-system can be described as a 6-tuple \( H = \{ P, C, t, t', \oplus, \otimes \} \) where \( P \) and \( C \) denote the plain-text space and the cipher-text space, respectively, whereas \( t \) and \( t' \) denote the encryption- and decryption-functions. \( \oplus \) and \( \otimes \) tag the two algebraic operations. Many different, yet limited homomorphic encryption schemes have been found. An overview of homomorphic encryption schemes that base on a single algebraic operation or a limited combination of two, is given in [6]. Advanced homomorphic schemes, such as [7] and [8], base on Gentry’s approach of bootstrapping a fully homomorphic from a somewhat homomorphic system [1] and provide addition and multiplication plus a normalization procedure that is supposed to allow unlimited chaining of operations in cipher-text space. This technique of reducing noise in the cipher-text space requires for an additional formal descriptive item, extending \( H_1 \) to \( H_2 = \{ P, C, t, t', \oplus, \otimes, r \} \), introducing a reduction-function \( r \), which takes a noisy cipher-text and transforms it into an equivalent with reduced noise.

Before we enter the detailed description of our concept, we want to demonstrate a very simple but incomplete (details below), algebraically homomorphic encryption scheme. However, our concept works with any encryption scheme in this class but this particular scheme is intended to give an idea of what an encrypted circuit representation, that we introduce in Section IV, may look like.

Let \( p \in \mathbb{N} \) be a large prime integer as a secret key and let the operands \( a \) and \( b \) be two arbitrary integers with \( (a, b) < p \in \mathbb{N} \). Then we can perform an encryption of \( a \) as \( a' = a + r \), where \( r < p \) is a large random integer. The cleartext \( a' \) can be calculated as the remainder of \( a' \mod p \). We can then perform an encrypted addition \( (a'+b') \) which extends to \( (a' + b') = (a + (r_1 \cdot p)) + (b + (r_2 \cdot p)) \) and \( a + b + (r_1 + r_2) \cdot p \) when decrypted \( mod p \) yields \( (a + b) \). The multiplication is performed as \( (a' \cdot b') = (a + (r_1 \cdot p)) \cdot (b + (r_2 \cdot p)) \) and \( a \cdot b + a \cdot (r_2 \cdot p) + b \cdot (r_1 \cdot p) + (r_1 \cdot r_2) \cdot p \cdot p \) mod \( p = (a \cdot b) \).

**Example 1.** \( a = 5, b = 4, p = 23, r_1 = 6, r_2 = 3; \)

\[
a' = 5 + (6 \cdot 23) = 143; b' = 4 + (3 \cdot 23) = 73;
\]

\[
a' + b' = 216, 216 \mod 23 = 9;
\]

\[
a' \cdot b' = 10439, 10439 \mod 23 = 20
\]

It is easy to prove that this scheme is incomplete, because if the result of an operation between the two operands \( a \) and \( b \) exceeds the prime modulus \( p \), the decryption fails. So starting with two clean plain-text items, the intermediate result grows towards the modulus with every operation and in this sense is polluted. To compensate for this, a fully homomorphic encryption scheme must define a normalization (Gentry calls this a reencryption procedure) of the intermediate result. In the case of the system shown here, a normalization would be any function that can minimize the remainder \( mod p \) of the result while preserving the parity \( mod p \). Gentry addresses this problem by generating a public key that contains a decryption hint. This hint allows to homomorphically decrypt the intermediate result in the encrypted domain, which means that the plain-text of the argument remains unknown. With the plain-text at hand in cipher-space, it is possible to reencrypt the plain-text which generates a new cipher of the plain-text with reduced noise.

### IV. Circuit Encryption

To encrypt a circuit we encode a bit in a cipher text’s property of having an even or odd remainder modulo a secret
prime key. This can be easily reduced to a boolean algebra.

**Definition 1.** In boolean expressions the operator \( \oplus \) denotes the XOR operation.

To model circuits, using the encryption scheme, introduced in Section III, we will be mapping 0-bits to even integers (resp. even remainders modulo a prime key) and 1-bits to odd integers. XOR-operations will be represented by the integer addition, while the integer multiplication represents a boolean AND-operation. This allows us to simulate chains of boolean operations by means of simple integer arithmetics.

**Example 2.** The boolean term \( r = (a \oplus b) \oplus (a \land b) \), which results in a boolean OR-operation, can be expressed in integer arithmetics as \( r = (a + b) + a \cdot b \), assuming \( a \) and \( b \) being even or odd integer representations of bit values.

**Definition 2.** We will be using the notation \( \circ \) for the composite OR-operation in integer arithmetics which is defined as \( a \circ b = (a + b) + (a \cdot b) \).

### A. Encrypted Memory Access

A basic circuit that implements read-access to memory is depicted in Figure 1. In that diagram, the memory values are drawn from static memory cells without further logic from the m-wires, which is a notation that closely relates to a software simulation. The output bit of this single bit memory-column with 2 address lines can be calculated as \( c = (\neg a_0 \land \neg a_1 \land m_0) \lor (a_0 \land a_1 \land m_1) \lor (\neg a_0 \land a_1 \land m_2) \lor (a_0 \land a_1 \land m_3) \).

\[
\begin{align*}
\text{row}_0 &= (a_0 + 1) \cdot (a_1 + 1) \cdot m_0, \\
\text{row}_1 &= (a_0 \cdot (a_1 + 1) \cdot m_1), \\
\text{row}_2 &= ((a_0 + 1) \cdot a_1 \cdot m_2), \\
\text{row}_3 &= (a_0 \cdot a_1 \cdot m_3), \\
c &= \text{row}_0 \circ \text{row}_1 \circ \text{row}_2 \circ \text{row}_3
\end{align*}
\]

**Example 3.** Let \( m_0..m_3 \) be the random bit memory column \{1, 0, 1, 0\} represented by the integer sequence \{5, 4, 9, 6\} and let \( a_0..a_1 \) be the sequence \{8, 3\} representing the decimal address 2 in binary form. Then the memory access described in integer arithmetic can be calculated as follows (please remember, that we are operating on a simplified representation model with a constant random \( r \) of 0):

\[
\begin{align*}
\text{row}_0 &= ((8 + 1) \cdot (3 + 1) \cdot 5) = 180 \\
\text{row}_1 &= (8 \cdot (3 + 1) \cdot 4) = 128 \\
\text{row}_2 &= ((8 + 1) \cdot 3 \cdot 9) = 243 \\
\text{row}_3 &= (8 \cdot 3 \cdot 6) = 144 \\
r &= 180 \circ 128 \circ 243 \circ 144 = 826087619 \equiv 1
\end{align*}
\]

Performing access with \( a_0..a_1 = \{5, 8\} \), representing the decimal address 1 in binary form yields the following calculation:

\[
\begin{align*}
\text{row}_0 &= ((5 + 1) \cdot (8 + 1) \cdot 5) = 270 \\
\text{row}_1 &= (5 \cdot (8 + 1) \cdot 4) = 180 \\
\text{row}_2 &= ((5 + 1) \cdot 8 \cdot 9) = 432 \\
\text{row}_3 &= (5 \cdot 8 \cdot 6) = 240 \\
c &= 270 \circ 180 \circ 432 \circ 240 = 5118619002 \equiv 0
\end{align*}
\]

We are able to access encrypted memory providing an encrypted address to the circuit such that the access procedure reveals neither memory address, nor memory content. Additionally we can observe that accessing memory with a different representation of an equivalent plain-text address results in a different representation of the accessed memory content. Also note the fact that we always have to solve the entire circuit, since we have no possibility to decide whether a particular row holds an encrypted 1 and therefore the calculation of the following row results can be omitted.

To assign a memory cell a new value, this new value representation is passed as \( i \) (input) to the write function. The writing-procedure then iterates over all memory rows and generates the row-select signal (which is an analogy to the select signal used in memory hardware) \( row \) as shown in Example 3 for \( \text{row}_0 \) to \( \text{row}_3 \). For each \( row \) we generate the new cell value as \( m_{\text{new}} = (\text{row} \land i) \lor (\neg \text{row} \land m) \).

This assigns the new value if the row is selected and the old value \( m \) otherwise. Note again, that even if not selected, every cell is assigned a new equivalent of the old representation. To implicitly decide between memory read- and write-access, we introduce an encrypted write-signal analogy that indicates the direction of the data flow. Implicit decision means, that there is, of course, no true decision, because the new value is a logical-numeric calculation over all given target bit representations (memory cells) and the selecting bit representations (address lines). The full-fledged bi-directional access function for a single bit column in the address-space of \( A \) reads as follows:
∀x ∈ A : m_x = (row_x ∧ write ∧ i) ∨ (row_x ∧ ¬write ∧ m_x) ∨ (¬row_x ∧ m_x), c = √(row_x ∧ m_x); x ∈ A, |x| = 1

Since the result of the access function is a logical combination of all memory cells, the complexity of memory access, which depends on the circuit size, is an almost linear function. The number of boolean gates that have to be processed in order to determine r during a read access is given as

\[ f(\text{size}) = (\text{size} * \neg) + (2 * \text{size} * \land) + ((\text{size} - 1) * \lor). \]

B. Encrypted Arithmetic-Logical Unit

To model an encrypted ALU, we apply a similar technique as we use to implement memory access. The ALU essentially consists of a couple of simple circuits that are applied constantly to the input signals a and b and, in every cycle, produce the operation-specific output. The command-switch \( o_0, o_1 \), which takes an opcode selects, the appropriate function result for the output wire and is in this sense equivalent to the address-selection in the memory circuit. Assuming the command-encoding for the selector lines \( \{o_0, o_1\} \) as

\[ \{0, 0\} \equiv add, \{1, 0\} \equiv and, \{0, 1\} \equiv xor, \{1, 1\} \equiv not \]

and the following series of equations which model the ALU functions in boolean logic

\[ c_{add} = \text{fulladder}(a, b), c_{and} = a \land b, c_{xor} = a \oplus b, c_{not} = \neg a \]

then this renders the result of the particular operation, denoted by \( o_0, o_1 \):

\[ c = (c_{add} \land (\neg o_0 \land \neg o_1)) \lor, (c_{and} \land (o_0 \land \neg o_1)) \lor, (c_{xor} \land (\neg o_0 \land o_1)) \lor, (c_{not} \land (o_0 \land 0_1)) \lor \]

Because the ALU function selection and memory access are comparable, we are not providing further details on ALU circuit modeling at this point. The transformation into integer arithmetic can be directly derived from the memory model.

C. Branching

Branching is the operation type that has the program counter as target register. We have unconditional branches (jumps) and those that depend on the system’s state. An unconditional jump is performed by copying the target address, provided by the jump command, to the program counter. The ability to perform dynamic branches, is one of the advantages of our concept, compared to other approaches. Actually, most conditional branches are directly influenced by a system state indicator, a flag. Flags offer the opportunity to steer the program flow according to events, like a comparison result (i.e. equality or zero) or a special arithmetic case (i.e. the carry value or the sign of an integer). In the following we assume a CPU wordsize of \( \eta \).

Let \( F \) be the set of flags, \( PC \) the program counter register, \( DR \) the data register and \( CR \) the command register. These latter two registers are loaded during the CPU fetch cycle with an operation code and an operand value. The functions \( jmp(CR), bcc(CR) \) and \( bz(CR) \) take the command register as input and return \( true \) (an odd representation) if the command register contains the respective command. \( jmp \) yields true if the loaded operation in the command register is an unconditional jump, \( bcc \) is true for a branch-if-carry-clear operation and \( bz \) is true for a branch-if-zero operation. The next address to be assigned to the \( PC \), following the program flow, is then

\[ \forall x : x \in \{0, (\eta - 1)\}, \]

\[ PC_x = (jmp(CR) \land DR_x) \lor (bcc(CR) \land DR_x \land \neg F_{carry}) \lor (bz(CR) \land DR_x \land F_{zero}) \lor (\neg jmp(CR) \land \neg bcc(CR) \land \neg bz(CR) \land (PC + 1)_x). \]

This applies a selection technique similar to the memory access function above. In this case, the \textit{memory values} are replaced by the two possible new addresses and the \textit{address selectors} are replaced by the command functions \( jmp \), \( bcc \) and \( bz \).

Termination

Due to the structure and the mathematical isolation of the processing model, the host has no possibility to decide, if and when the execution of the encrypted program ends. The encrypted program, being locked up in the cipher-space, can neither generate any unencrypted signal that can be evaluated from outside the container, nor can it manipulate a resource (memory locations or flags) that can be distinguished from other encrypted resources from the outside. The encrypted program has no influence on the execution timing and is completely passive. To practically address this particular termination problem, we propose the definition of the number of cycles, that have to be performed to safely execute the encrypted program. A proof under what conditions this problem can be solved remains to be given.

V. Cloud Applications

One of the major problems in Cloud Computing is the fact that the security and confidentiality of a remote resource cannot be technically validated by the customer. As a consequence, the security aspect is still subject to contracts and trust. In contrast to the execution of programs, the transmission of data can be well protected by different cryptographic techniques. Asymmetric methods can satisfy a number of requirements in the field of encrypted communication, such as confidentiality, integrity and authenticity. The system that we present in this paper can act as a basis for protocols and methods that strengthen the security of program execution in distributed environments. In this Section we describe four example use cases.

A. Delegation

The delegation of computation to a remote resource is the most obvious use case for our system. Many different delegation scenarios exist today. They range from the provision of machines and platforms through resource providers to
remote data storage and remote (web-) desktop applications, like Google Apps, the Zoho office suite and others. Whatever granularity any of these services have, they all share the common danger of being physically located outside the jurisdiction of the customer. To address this problem, the customer generates an encrypted version of his software and data and transmits the resulting aggregate to the resource. The program included in the encrypted package is then executed on the remote resource in encrypted form and after the program is finished, the entire encrypted package is transferred back to the customer, who recovers the result by decrypting designated parts of the package.

B. Remote Search

A desirable feature of a search engine is an encrypted search function. The customer provides an encrypted search argument to the search engine and receives an encrypted result. The search engine itself can neither determine argument nor result of the search operation. The encrypted search can be implemented in different ways, concerning the encryption of the database that contains the data to be searched. One option is the encryption of the entire database with the customer’s key, which implies that the data is only accessible by one customer. Another option is an encrypted search function that handles plaintext input data. The search engine subsequently inserts data into the search function and thus generates the encrypted result which is then transmitted to the customer.

C. Mobile Code

A special case of delegation is mobile code, as found in different software agent systems. In our case, mobile code is an encrypted aggregate of software and data that can be passed from one resource to another. The purpose might be the collection of data, if the customer (the information recipient) is only interested in the final result, after different resources have provided their input to the aggregate, or after the aggregate has been circulating for a defined period of time. Each information provider inserts its data and executes the encrypted import function which calculates a new intermediate result and disguises the last input. On the one hand the intermediate results cannot be evaluated by the information providers, on the other hand the information recipient, though being able to decrypt the final result, cannot evaluate the different input data.

D. Multi-Party Computation

The classic multi-party (MP) computation describes a scenario where two parties want to execute a function with secret input data on each side and a common output. This can be achieved with our system as sketched in the Mobile Code use case. In the MP scenario all participants act as both information provider and information recipient. An advantage of our system over the approach of the garbled circuits [2] is that the garbled circuits require much more communication between the interacting parties to generate and exchange the function representation.

VI. IMPLEMENTATION

We provide prototype- implementations of our concept in Java and C. This Section describes the software-architecture and the system properties. We also want to show that because of its simple design it can be easily integrated into existing applications for rapid prototyping or be extended to meet special requirements.

A. Architecture

The layered software architecture of our system is depicted in Figure 2. We use a Java VM as a base, upon which we built the execution engine with a dependency on an implementation of homomorphic encryption as a library.

![Software Layers](image)

The encrypted program itself is independent of the particular type of encryption, as long as the assembler module and the execution engine use the same encryption library. To enable a pluggable architecture, we have defined a narrow interface that ensures interoperability with any suitable and available encryption library. The library has to provide appropriate functions like encode/decode/recode and the two arithmetic functions.

The encode-function takes an integer and generates an array of encrypted bit representations depending on the secret key. The decode-function takes an array of encrypted bit representations and decodes these, applying the secret key. The recode-function is the normalization-procedure and takes a public key to produce a representation of an encrypted bit that contains less noise. The modular system design allows for quick replacement of the crypto-library, in case a faster implementation becomes available.

B. System Properties

The prototype- implementations apply a memory word length of 13 bits in little-endian format. A word contains eight bits of data in the data compartment (bits 0 to 7) and a five bits wide command (bits 8-12). This enables our sample processor architecture that bases on a memory-access principle, where command and data can be fetched in the same instruction cycle. If a memory location is written to, only the data compartment is modified, whereas the command compartment remains untouched. The memory circuit contains a total of 256 words that may be randomly accessed.

The instruction set implements the two addressing modes immediate and direct with commands in different classes like memory access (load, store), program flow (comparison, jumps...
and branches), logic functions (and, or, xor) and arithmetic functions (add, rotate, shift). Immediately addressed operands are contained in the command-word, while directly addressed operands are described by their memory location. The reference Java-implementation contains a symbolic assembler, that is capable of generating encrypted machine code based on a cryptographic library, as depicted in the architectural overview.

C. Test Setup

Our test suite consists of a 2.4GHz Intel Core 2 Duo with 4GB of 667MHz DDR2 SDRAM platform and Java 5. The average cycle duration on our test machine is 3ms. A cycle processes 129397 gates (68933 AND, 44988 XOR, 15476 NOT), the C implementation is slightly faster. Since these figures describe the simplified crypto-system, they only contain the plain time consumption of the circuit evaluation. To be able to make a statement about a production level configuration, where a sound encryption scheme is applied, these figures may serve as a basis for a rough runtime estimation of the program itself.

VII. Open Issues and Future Work

We have implemented the Smart-Gentry crypto system [8] and are currently integrating it into our approach. Performance figures should be available soon. One of the main issues of our concept is the termination problem. To solve this problem, a crypto-system that can selectively decrypt information is required. There are a number of possible applications for our concept, including a couple of older problems, as shown in Section V. The environment presented in this paper has limited capabilities and a dependency between memory size and performance, which makes it primarily suitable for small problem sizes. By extending the capabilities of our concept to interact with the host system, we will be able to perform calculations on portions of secret data or secret algorithms, that are part of a larger system. Currently, we are able to inject data into the encrypted environment, which is sufficient to receive process data from outside the cipher-space. However, this induces further problems, like the correctness and consistency of the encrypted code and data. An obvious field of application is correctness and consistency of the encrypted code and data. In this paper we presented a method to perform the encryption of encrypted programs operating on encrypted data. In contrast to other solutions, the code as well as the processed data is held entirely in the cipher-space, but still remains dynamic and can be provided with multiple process data after having been transferred to the executing host. We have described a simple homomorphic encryption scheme as a reference model and developed a method to represent circuits by means of homomorphically encrypted integer arithmetics. Applying the basic logic function representations, we sketched how to build different microprocessor primitives like memory-access logic and arithmetic operations. We then developed a simple CPU and system model and presented the reference implementation of our model. We provided basic performance figures that help to establish a rough runtime estimation for encrypted programs. We discussed different example use cases for our concept in the area of distributed computing and sketched how the mentioned problems can be addressed applying our system.

REFERENCES