2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications

How Practical is Homomorphically Encrypted Program Execution? An Implementation and Performance Evaluation

Michael Brenner, Henning Perl and Matthew Smith
Distributed Computing Security Group, Leibniz Universität Hannover, Germany
{brenner,perl,smith}@dcsec.uni-hannover.de

***Abstract*—Homomorphic cryptography has received a lot of attention due to potentially ground breaking advances in cryptography. However it is also surrounded by a lot of hyperbole such as "ground breaking advances", "this will solve all Cloud computing problems" to "it is completely impractical" and "it will never work for real world problems". In previous work we showed how homomorphic encryption can be used to execute arbitrary programs in encrypted space, showing that at least in theory real world problems can be computed protected by homomorphic cryptography without losing generality. In this paper we expand our work to evaluate how practical current homomorphic cryptography based on the Smart-Vercauteren system is for executing arbitrary programs on untrusted resources. For this we present the implementation of a method to compute non-linear secret programs on an untrusted resource using encrypted circuits embedded in an encrypted virtual machine. We successively show how a processor architecture using encrypted circuits can be implemented so it can support read and write memory access, dynamic parameters and non-linear programs that render branch-decisions at runtime. The system comprises the runtime environment for program execution and an assembler to generate the encrypted machine code. We present performance evaluation of the sub-components as well as the complete system. The system represents a flexible prototype for homomorphic program execution in software and system architecture.**

I. Introduction

Remote execution of programs and remote data storage are important aspects of todays IT landscape, in particular due to the boom of Cloud computing. During transport and storage both programs and data can be stored in encrypted form to prevent theft and/or manipulation. However, during execution both the program and data must be decrypted. This means, that programs, algorithms and data cannot be properly protected when executed on a remote resource and are entirely under control of the resource owner. Recent fully homomorphic encryption schemes address the problem of computing on encrypted data by providing addition and multiplication of encrypted bit representations. This allows for a theoretical approach to encrypted arithmetics with arbitrary precision when operating on composite numbers that consist of columns of such encrypted bits. Also circuits can be represented when mapping the homomorphic operations to boolean gate functions. Even though it is well known, that integer arithmetics can simulate boolean circuits (Gentry explicitly mentioned this fact in [5]) most approaches to transform algorithms into circuits only encrypt the data and model the program flow in a sequence of native commands for the target platform. This means, that the algorithm itself is disclosed. Furthermore the existing frameworks that generate encrypted circuits, like [12] can only assemble linear circuits that are executed in one single pass. Recent approaches to combine garbled circuits with homomorphic cryptography [6] do not sufficiently address this problem. This paper puts an emphasis on investigating the practical potential of a universal program execution model rather than minimized and specialized circuits. One of our main aims in this paper is to put some concrete performance context to the hype surrounding arbitrary secret program execution which is sometimes touted as the silver bullet for Cloud computing security. In previous work [2] we showed how in theory arbitrary program executing is possible and sufficiently expressive to be useful for real world problems. Here we expand on this work showing how these building blocks can be implemented and put together using the fully-homomorphic Smart-Vercauteren crypto-system to create an encrypted virtual machine that is able to execute arbitrary code for the defined machine model. We evaluate this concrete implementation and show the current performance limitations. The paper structure is as follows: related work and the state-of-the-art is discussed in section II. Section III introduces the approach of encrypting circuits using homomorphically encrypted bits and gate representations. We describe the processor primitives and discuss details that have to be considered for implementation. Some details of our working implementation of the Smart-Vercauteren cryptosystem are presented in Section IV. A brief overview of open issues and future work is given in Section V. Section VI concludes the paper.

978-0-7695-4745-9/12 $26.00 © 2012 IEEE
DOI 10.1109/TrustCom.2012.174

II. Related Work

Since the breakthrough work of Gentry [5], a number of similar approaches to fully homomorphic encryption appeared, like Smart et al. [15] or van Dijk et al. [17], that share the basic idea of bootstrapping a *fully* homomorphic encryption (FHE) scheme from a depth-limited *somewhat* homomorphic scheme (SHE). Due to the computational overhead of current FHE schemes, the question arises, if the underlying SHE schemes can also be used for more practical homomorphic encryption. Recent proposals, like Lauter et al. [13] follow this approach. However, the fully homomorphic encryption is still subject to progress in terms of new considerations of hardness assumptions (Stéhle et al. [16]), conceptual simplicity (Coron et al. [3]) or yet different mathematical basics (Brakerski et al. [1]).

There are different paradigms for secure delegation of computation like *secure function evaluation* (SFE) mostly based on *Yao's Garbled Circuits* [18] and extensions by Malkhi et al. [12] or Kolesnikov et al. [11]. Garbled circuits have also been combined with homomorphic encryption by Kolesnikov et al. [10] and Gentry et al. [6] to overcome their inherent disadvantage of of being limited to static one-pass boolean circuits.

A theoretical approach to achieve privacy of memory access patterns and algorithm execution in a special type of Turing Machines is the Oblivious Random Access Machine (ORAM) by Goldreich et al. [7] [8]. There are recent proposals to reduce the complexity of ORAMs by Pinkas et al. [14] and further developments towards practical applications by Damgård et al. [4] and Goodrich et al. [9].

III. Construction of an Encrypted System

Our objective is to construct an encrypted machine model which has the same properties as a conventional microprocessor. In particular, the encrypted CPU is required to hide the confidential functionality of the executed software. Additionally, the CPU must be able to handle any program flow, including loops and branches. That means, that no loop-unfolding or code multiplication for case handling is needed. Also important to mention is the demand to provide programmers with an environment that a priori is encrypted and does not require crypto-API calls entangled in business logic to provide security.

We will develop the machine step-by-step by first defining the building blocks of the model. We will then identify the interfaces needed for efficient operation between the components and eventually assemble the pieces to build the target system. For every component we provide performance figures for different encryption key sizes and model parameters. The security parameter λ in the performance tables and other related parameters of the Smart-Vercauteren key generator are described in Section IV.

Notation. For the definition of interfaces we use the following notation: characters (a) and strings (*read*) denote a single bit value whereas overlined characters ($\bar{a}$) and strings ($\overline{reg}$) denote a bitvector. Component names come as strings in caps.

The construction of the system is divided into three main blocks:

- Memory access
- Arithmetic-logical unit
- Control unit

As we will show, all of these components can be implemented applying binary demultiplexers (short *demux*). A demux is essentially a binary selector where the request contains the n-bit address of an item in a column of 2^n items.

A. Encrypted Memory Access

Memory access is one of the key components of the encrypted system. This is the part of the system that takes most of the evaluation time in the sequential circuit simulation. In fact, the evaluation time is at least[1] linear in the size of the memory. The output bit of this single bit memory-column with two address lines can be calculated as

$$\begin{aligned} c \;=\; & ((a_0+1')\cdot(a_1+1')\cdot m_0)\circ(a_0\cdot(a_1+1')\cdot m_1) \\ & \circ((a_0+1')\cdot a_1\cdot m_2)\circ(a_0\cdot a_1\cdot m_3) \end{aligned} \tag{1}$$

Notation. For the transformation we replace the bits by encrypted representations and the boolean operations AND $\wedge$ and XOR $\oplus$ by their arithmetic equivalents mod 2 (multiplication and addition). The boolean OR $\vee$ is replaced by the composite operation $(a+b)+(a\cdot b)$ and denoted by $\circ$. The addition of an encrypted $1'$ is equivalent to the boolean negation.

To access the memory, we iterate over all memory addresses and evaluate the switching function (1). A term of the form $(a_0+1')\cdot(a_1+1')$ in Eq. 1 is equivalent to a *row select signal* in DRAM technology. It results in a 1 if the current row is selected by the request, 0 otherwise and multiplied with the memory content of the row yields the result for that row. Since the circuit selects at most one address at a time, we can

[1] the number of required gates is linear in the size of the memory, whereas the simulation time for larger circuits increases polynomially due to the underlying encryption (see Table I)

simplify the function by replacing the OR-operations by XORs.

With this circuit 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. Assuming that a probabilistic cryptosystem provides different cipher representations for equivalent plaintext, we can observe that accessing memory with a different representation of an equivalent plaintext address results in a different cipher representation of the accessed memory content.

To assign a memory cell m a new value during a write operation, this new value representation is passed in a register *reg* to the access function. For each memory row we generate the new cell value as $m_{new} = (row \wedge reg) \vee (\neg row \wedge m)$ with *row* being the row select signal as shown above. This assigns the new bit value *reg*, if the row is selected (and thus has the value 1) and the old value m otherwise. The universal memory access function for the address space $\mathfrak{A}$ reads

$$\begin{aligned} \forall a \in \mathfrak{A} : m_a \quad &\leftarrow \quad (row_a \wedge write \wedge reg) \vee \\ &\qquad (row_a \wedge read \wedge m_a) \vee \\ &\qquad (\neg row_a \wedge m_a) \\ reg \quad &\leftarrow \quad (row_a \wedge read \wedge m_a) \vee \\ &\qquad (row_a \wedge write \wedge reg) \vee \\ &\qquad (\neg row_a \wedge reg) \end{aligned} \tag{2}$$

Table I shows the performance figures of our implementation of the access function. The time is measured as $\frac{\Delta clock()}{CLOCKS\_PER\_SEC}$ using the standard functions and macros from clib. It turns out, that write access is much faster than read access for larger memory arrays, because the noise of the memory bits accumulates in the target register and thus a lot of recrypting is required. Also notice, that early reads are cheaper than reads when the system is *in tune*, i.e. when all of the bits in the system have a comparable noise.

Memory rows	$\lambda = 384$	$\lambda = 512$	$\lambda = 768$	$\lambda = 1024$
8	1 / <1	1 / <1	1 / <1	1 / <1
16	4 / <1	4 / <1	5 / <1	5 / <1
32	17 / <1	19 / <1	21 / <1	25 / <1
64	9-18 / 4	9-19 / 4	11-22 / 5	13-25 / 6
128	21-63 / 9	22-68 / 9	25-76 / 11	30-90 / 12
256	51 / 15	53 / 15	62 / 17	71 / 21

Table I
TIMINGS FOR MEMORY ACCESS IN SEC. (READ/WRITE)

The interface for the memory access function MA is

$$\overline{reg} \leftarrow MA(\overline{adr}, \overline{reg}, read)$$

where *reg* is a register, *adr* is the accessed memory address and *read* is the data direction flag. The function calls to MA in the control unit do not need to hide the data direction, because it is obvious. In light of this, we have implemented the memory access over a memory array `A` according to Eq. 2 in two slightly modified functions as follows:

```
reg MAr(adr)
{
  for(i = 0..ROWS)
  {
    sel = ROWSELECT(i,adr);
    for(j = 0..WORDSIZE)
    {
      reg.j  = (A[i].j * sel)+(reg.j  * !sel);
    }
  }
}

MAw(adr,reg)
{
  for(i = 0..ROWS)
  {
    sel = ROWSELECT(i,adr);
    for(j = 0..WORDSIZE)
    {
      A[i].j = ( reg.j * sel)+(A[i].j * !sel);
    }
  }
}
```

where | denotes the composite OR-operation and $X.y$ represents the y-th bit of the word X. The function *ROWSELECT* results in a logic 1, if the encrypted binary representation of the accessed memory row *adr* equals the row-iterator *i*:

```
sel ROWSELECT(i,adr)
{
  sel=ENCRYPT(1);
  for(j = 0..ADRSIZE)
  {
    if(int.j==1)
      temp=ENCRYPT(1)*adr[j];
    else
      temp=ENCRYPT(1)*!adr[j];

    sel=sel*temp;
  }
}
```

B. Encrypted Arithmetic-Logical Unit

The encrypted ALU can also be constructed applying demux circuits. Like an address in the memory model, the requestor provides an opcode which selects the function to be performed over the two operands. The simplified function that renders the result of an ALU operation $\{o_0, o_1\}$ over the function set $\{c_{and}, c_{add}, c_{or}, c_{xor}\}$ is

$$c = (c_{or} \wedge (\neg o_0 \wedge \neg o_1)) \oplus (c_{xor} \wedge (o_0 \wedge \neg o_1)) \oplus (c_{and} \wedge (\neg o_0 \wedge o_1)) \oplus (c_{add} \wedge (o_0 \wedge o_1))$$

The ALU is modeled to operate on 1-bit operands. To process machine words of n bits, we couple n instances of the circuit. Since we have an operation with dependencies between adjacent operand bits (the addition), we need to provide state information (the carry) from one ALU stage to the next one. This leads to the first basic interface $(res, carry) \leftarrow ALU1(\overline{opcode}, op1, op2, carry)$. This allows us to implement an n-bit ALU with a simple ripple-carry adder. There are much more efficient adder variants, but on the other hand these adders need much more gates to implement. In a hardware design, the gates of a circuit are switched almost in parallel, but it's worth to recall that we work with a sequential circuit simulation. That means that the time to evaluate a circuit depends on the number of gates.

The ripple carry n-bit ALU with the extended interface $(\overline{res}, carry) \leftarrow ALU(\overline{opcode}, \overline{op1}, \overline{op2}, carry)$ can be easily realized by iterating with the $ALU1$ implementation over the operands from LSB to MSB.

```
(res,carry) ALU(opcode,op1,op2,carry)
{
  for(i = 0..n) //little endian operands
  {
    (res[i],carry)=ALU1(opcode,op1[i],op2[i],carry);
  }
}
```

The performance figures for the ALU implementation are depicted in Table II.

Operand size	$\lambda = 384$	$\lambda = 512$	$\lambda = 768$	$\lambda = 1024$
8 bits	0.07 s	0.08 s	0.09 s	0.1s
16 bits	0.14 s	0.16 s	0.19 s	0.22 s
32 bits	0.28 s	0.32 s	0.38 s	0.45 s

Table II
ALU TIMINGS

C. Encrypted Branching (the control unit)

The control unit is the most important part of our implementation. It is the *glue* in the system which interconnects the other components and controls data and program flow. The CU is a state machine and itself is controlled by the program code which contains opcodes and operands. The CU reads a memory word and determines the function it is required to perform by passing the opcode to the decoder unit. The decoder *parses* the opcode and activates the corresponding functional circuits that render the correct output. This output is twofold: on the one hand the CU generates an arithmetic or logic result of an operation over one or two operands, on the other hand it computes the memory address of the next command to be executed. This program flow is controlled by the state of the machine, specifically the accumulator register, the program counter and the flags: alternative program flow and decisions are typically determined according to events like an addition carry of a zero-result of an arithmetic operation.

```
CONTROL()
{
register ac = 0; //accumulator
register pc = 0; //program counter
flag carry = 0;
flag zero = 0;
extern flag brk = 0;

while(!brk)
{
  temp = MAr(pc); //immediate addressing
  opi = GETOPERAND(temp);
  cmd = GETOPCODE(temp);

  temp2 = MAr(opi); //absolute addressing
  opa = GETOPERAND(temp2);

  //ALU operations
  (ac_new_i,carry_new_i) = ALU(0,ac,opi); //imm
  (ac_new_a,carry_new_a) = ALU(0,ac,opa); //abs

  //LOAD
  ac_new_li = opi; //imm
  ac_new_la = opa; //abs

  //STORE
  mem_new = ((ac & cmd==ST) | (opa & cmd!=ST))
  MAw(opi,mem_new);

  //UPDATE AC
  ac = (ac_new_i & cmd in{OR,XOR,AND,ADD}) |
       (ac_new_a &cmd in{ORa,XORa,ANDa,ADDa} |
       (ac_new_li & cmd==L) |
       (ac_new_la & cmd==La);

  //UPDATE FLAGS
  carry = (carry_new_i & cmd==ADD) |
          (carry_new_a & cmd==ADDa);
  zero = (zero_new_i & cmd in{{OR,XOR,AND,ADD}) |
         (zero_new_a & cmd in{ORa,XORa,ANDa,ADDa})|
         (zero_new_li & cmd==L)|
         (zero_new_la & cmd==La);

  //JUMP & BRANCH
  pc = ((opi & cmd==JMP) |
       (opi & cmd==BCC && carry==0) |
       (opi & cmd==BZ && zero==1) |
       (pc+1 & cmd notin{JMP,BCC,BZ}));
}
}
```

The control unit uses the auxiliary circuits $GETOPERAND(word)$ and $GETOPCODE(word)$ to divide a memory word into data and an opcode. This combined von-Neuman/Harvard architecture makes it easy to load a complete command in a single memory access As shown in the pseudo-code of the control unit, every machine cycle accesses the memory array 3 times to achieve obliviousness in the control flow:

- the first operation reads the word *opi* at the program counter *pc*
- the operand of the fetched word is a potential address, so the content of this address is also fetched as *opa*

Opcode	*Mnemonic*	*Description*
0/ 8	OR / ORa	logical or
1/ 9	XOR / XORa	logical xor
2 / a	AND / ANDa	logical and
3 / b	ADD / ADDa	logical add
4 / c	L / La	load register
5	ST	store register
6	JMP	unconditional jump
7	BCC	branch if carry clear
d	BZ	branch if zero

Table III
INSTRUCTION SET

Memory rows	$\lambda = 384$	$\lambda = 512$	$\lambda = 768$	$\lambda = 1024$
8	5 s	5 s	6 s	7 s
16	11 s	12 s	14 s	16 s
32	39 s	42 s	48 s	55 s
64	35 s	37 s	42 s	49 s
128	97 s	104 s	117 s	137 s
256	119 s	123 s	144 s	166 s

Table IV
CPU TIMINGS (ONE MACHINE CYCLE)

- the *opi* operand is also a potential store address for the *reg* or the *opa* value

The performance of the control unit is the sum of all components that get involved in a machine cycle. The timing figures for one cycle are depicted in the short Table IV.

IV. SMART-VERCAUTEREN HOMOMORPHIC LIBRARY IMPLEMENTATION

In this section we present our implementation of the fully homomorphic Smart and Vercauteren cryptosystem which we use as a basis for our approach. We needed to make some small modifications since our original implementation based on [15] and [5] was not stable enough for arbitrarily deep circuits and produced false results. To enable correct operation we modified the recrypt operation, specifically we improved the circuit to calculate the Hamming weights to make it shallower. However, this modification does not weaken the security of the cryptosystem compared to the original, because the recrypt operates under the public key only and thus cannot do other things, *any* attacker in possession of the public key could also do. The following Section presents the details of our implementation of the Smart-Vercauteren system.

A. Somewhat Homomorphic Scheme

In this subsection we present a somewhat homomorphic encryption scheme as defined by Smart et al. Decryption works only as long as the cypher text noise, more precisely the coefficients of the $C(x)$, is within certain bounds r. These bounds depend mainly on the parameter N. For our implementation and choice of parameters we get $r = 2^{\nu}/(2\sqrt{N}) = \frac{2^{384}}{16}$, see [15] for a more generic calculation.

Besides this restriction, the general idea is the same as with the fully homomorphic scheme. $G(x)$ generates an ideal $\mathfrak{p} = (G(x))$ in $\mathbb{Z}[x]/F[x]$, along with the two element description $\mathfrak{p} = \langle p, x - \alpha \rangle$ (which is used in the encrypt function). This gives us the homomorphism

$$\begin{aligned} \varphi_{\mathfrak{p}_i} : \mathbb{Z}[x] &\to (\mathbb{Z}[x]/F(x))/\mathfrak{p} \\ C(x) &\mapsto C(\alpha) \bmod p \end{aligned} \tag{3}$$

which we will use for the encryption.

1) Key generation:

Notation: We write polynomials in uppercase roman letters. For a given polynomial $G(x)$ of degree n, $(g_0, \ldots, g_n)$ denote the coefficients so that $G(x) = \sum_{i=0}^{n} g_i x^i$.

```
1: keygen(pk, sk)
2: {
3:     F(x) = x^n + 1        // monic, irreducible of degree n
4:     do {
5:         G(x) = random polynomial in B^even_{∞,n}(μ)
6:         ++ g_0            // make the constant coefficient odd
7:         p = fmpz_poly_resultant(G(x), F(x))
8:     } while p is not prime
9:     D(x) = F_mpz_mod_poly_gcd_euclidean(G(x), F(x))
10:    α = −d_0                          // α = root of D(x)
11:    (r, Z(x), t) = fmpz_poly_xgcd(G(x), F(x))
12:    pk.p = p; pk.α = α      // pk, sk are simply structs
13:    sk.p = p; sk.B = z_0 mod 2p
14: }
```

Discussion:

l. 3 $x^n + 1$ is always a safe choice for a monic, irreducible polynomial of degree n as it has no roots in $\mathbb{Z}[x]$.

l. 5 [15] defines

$$\mathcal{B}_{\infty,n}(r) := \left\{ \sum_{i=0}^{n-1} a_i x^i : a_i \in [-r, r] \right\} \quad .$$

Analogously, we define

$$\mathcal{B}^{\text{even}}_{\infty,n}(r) := \left\{ \sum_{i=0}^{n-1} 2a_i x^i : a_i \in \left[-\frac{r}{2}, \frac{r}{2} \right] \right\} \quad .$$

In the algorithm, we double randomly choose coefficients in $[-\frac{\mu}{2}, \frac{\mu}{2}]$.

l. 8 the Miller-Rabin prime number test is used here. $p \neq 0$ prime implies that $F(x)$ and $G(x)$ are coprime and further that $G(x)$ is irreducible ($F(x)$ is irreducible). $G(x)$ generates a principal ideal $\mathfrak{p}$ in $\mathbb{Z}[x]/F(x)$.

l. 9 the GCD-algorithm works on polynomials modulo p. We find the two element representation $\langle p, x - \alpha \rangle$ of $\mathfrak{p}$ with p being the norm

of $\mathfrak{p}$ and α a root of $F(x)$ mod p. The root of $D(x)$ is also a root of $F(x)$ and $G(x)$.

l. 11 the ext. GCD-algorithm sets the output so that $Z(x) \cdot G(x) = p \bmod F(x)$. Here we generate the subsecret key. The decryption algorithm requires us to calculate $\frac{1}{G(x)} = \frac{Z(x)}{p}$. We only need to round to the nearest integer, so only z_0 is relevant here as the subsecret key.

2) Encrypt, Decrypt, Add, Mult: The code for encrypt, decrypt, add and mult largely resembles the pseudocode given in [15] as all of the operations translate well to GMP calls.

In the encryption function we generate a polynomial $C(x)$ with the parity of the constant coefficient depending on the message to be encrypted. Then we use eq. 3 to transform the polynomial to the crypto space by evaluating $C(\alpha)$ mod p with fmpz_poly_evaluate().

After an addition, if the input ciphertexts were bound by b_1 and b_2, the result will be bound by $b_1 + b_2$ (the coefficients simply add up). After a multiplication however, the result will be bound by $b_1 + b_2 + b_1 \cdot b_2$. In the following we will examine how many multiplications are possible while the cipher text is still decryptable. After d multiplications, the output will be bound by $\mu^{2^d} = 4^{2^d}$.

$$4^{2^d} = \frac{2^{384}}{16} \Leftrightarrow 2^d = 190 \Leftrightarrow d = \lfloor log_2 190 \rfloor = 7 \qquad (4)$$

This will be enough for our recrypt() to work as well as leaving some operations for homomorphic gates.

B. Fully Homomorphic Scheme

In this section we present a version of the decryption algorithm consisting only of xor and and-gates. This allows us to apply the decryption to a ciphertext, returning a cleaner version. For this recrypt to be effective, the depth of the circuit has to be shallow enough. We present a minimal rounding function to account for this.

1) Key generation revised: In addition to B, α and p constructed in the somewhat homomorphic version of keygen, we now also have to construct a hint $\{\mathfrak{c}_i, B_i\}_{i=1}^{s_1}$ so that $\sum_{i=1}^{s_1} \mathsf{decrypt}(\mathfrak{c}_i) B_i = B$. Since we rely on hiding the value of B in the array B_i the security of this hint depends on the size of B_i. The security of the hint can be reduced to the *Subset-Sum* problem.

```
keygen()                                   // continued ...
B* = ⌊B/S2⌋                                // Step 1: Distribute
for (i = 0; i < S2; i++) {
    pk.B_i = B*
    pk.c_i = 1
}
for (i = S2; i < S1; i++) {
    pk.B_i = random in [−p, p]
    pk.c_i = 0
}
for (i = 0; i < S2; i++) {                 // Step 2: Add/Sub at random
    r+ = random in [0, p − 1]
    r− = −r+
    add and subtract r+ and r− from random B_j
}
for (i = 0; i < S1; i++ {                  // Step 3: Shuffle array
    j = random in [0, S1 − 1]
    swap pk.B_i and pk.B_j
    swap pk.c_i and pk.c_j
}
```

$pk.B$	$\lfloor \frac{B}{4} \rfloor$	...	$\lfloor \frac{B}{4} \rfloor$	$B - 4\lfloor \frac{B}{4} \rfloor$	rand	...	rand
$pk.\mathfrak{c}$	0		0	0	1		1

Table V
INITIAL DISTRIBUTION OF THE HINT

Discussion:

Step 1 In the distribution step the array is initialized as shown in table V.

Step 2 Addition and subtraction of random values only operate on the first S_2 (= 5). In each step the invariant $\sum_{i=1}^{s_1} \mathsf{decrypt}(\mathfrak{c}_i) B_i = B$ holds.

Step 3 In order to randomize the hint, we finally shuffle the array.

2) Recrypt: The purpose of the recrypt() function is to generate a *c*leaner ciphertext from a *d*irty one with the same cleartext value. The homomorphic operations add and mult add to the dirtyness of the ciphertext, making it eventually unrecoverable. Therefore, without a recrypt() function, we can only compute circuits of a fixed depth. Beyond that, the results are not correctly decryptable. The general idea of this function is to decrypt the ciphertext in the cryptospace using homomorphic operations. Recall that the decrypt function computes

$$\left(c - \left\lfloor B\frac{c}{p} \right\rceil\right) \equiv_2 \left(c - \left\lfloor \sum_i \mathfrak{c}_i B_i \frac{c}{p} \right\rceil\right) \equiv_2 \left(c - \left\lfloor \sum_i \mathfrak{c}_i (B_i \cdot c \bmod 2p)/p \right\rceil\right)$$

The right part of the equation we can compute using only the public key and the hint constructed in subsection IV-B1. But since the $\mathfrak{c}_i$'s are encrypted, the final output will also still be encrypted. The result is a cleaner encryption of c.

```
for (i = 0; i < S1; i++) {
    d = (B_i · c mod 2p)        // Note that d ∈ [0, 2)
```

```
3:     for (j = 0; j < T; j++) {
4:         C_ij = encrypt(⌊d⌋) · c_i mod p
5:         d = (d − ⌊d⌋) · 2          // conv. frac. base 2
6:     }
7: }
```

After running the algorithm above, each row of the matrix (C_{ij}) holds a binary representation of $(B_i \cdot c \bmod 2p)$ with $T-1$ bits of precision in the ciphertext. As an optimization in line 4 we can check wether $\lfloor d \rfloor$ is zero and then use either $\mathfrak{c}_i$ or encrypt(0). This saves an additional multiplication and provides a cleaner ciphertext matrix to start with. Next, we sum up individual rows with a circuit as shallow as possible. Therefore, the addition of the rows is split up in separate steps:

1) Calculate Hamming weights of the rows using symmetric polynomials as suggested by [15]. With this approach we get a much shallower circuit cmp. using half- and full-adders, since the poly $e_k(X_1, \ldots, X_n) = \sum_{1 \le j_1 < j_2 < \ldots < j_k \le n} X_{j_1} \ldots X_{j_k}$ directly gives the k-th bit of the Hamming weight of the input.
2) Shift and merge the Hamming weights according to significance of the column. Note that we don't need any circuit gates here as we just *r*ewire.
3) Apply carry-save adder to three rows at a time until there are only two rows left. This allows for a constant circuit depth.
4) Finally, do one ripple-carry addition which has a linear circuit depth with respect to the length of the rows.

After this last step, we have reduced the matrix to one column holding a fixed precision floating point representation of $\sum_i \mathfrak{c}_i (B_i \cdot c \bmod 2p)/p$.

3) A shallower rounding function: To produce the cleaner encryption of the input, [15] suggests a rounding using the last two floating point bits. However in our implementation this resulted in a circuit that was overall too deep to compute the correct value.

We therefore suggest an alternative rounding using just the last bit. By doing so we are saving additional circuit depth while still being able to correctly "clean" the value. Effectively we are able to reduce the depth by two multiplications.
The output of tour rounding is then given by

$$\left(c - \left\lfloor \sum_i \mathfrak{c}_i (B_i \cdot c \bmod 2p)/p \right\rceil \right) = (c + e_0 + e_1) \bmod 2 \quad .$$

C. Implementation parameters

For the implementation of our system[2], we used different security parameters λ. It defines the cipher size and is the upper bound μ for picking a random polynomial in the Smart-Vercauteren key generator. The public key contains a decryption hint that is split into $S2$ addends and randomly distributed over an array of size $S1$. We use a constant configuration of $S1 = 8$ and $S2 = 5$ for all experiments which leads to the key file sizes summarized in Table VI.

key	$\lambda = 384$	$\lambda = 512$	$\lambda = 768$	$\lambda = 1024$
secret	1.8	2.5	3.7	4.9
public	16.7	22.3	33.3	44.5

Table VI
KEY FILE SIZES (KB)

To track the cumulative noise of ciphertexts and to efficiently trigger the recrypt operation, we attach a numeric noise measure to every encrypted bit. A multiplication result (an AND operation) inherits the sum of the two operand's noises, additions (XORs) are free in our model. Before a multiplication the sum of the two noises is checked and if the sum exceeds the value 7 (see Eq. 4) then the *noisier* value is recrypted.

V. Open Issues & Future Work

This implementation is still at an early stage, however, there are many possible applications, including a couple of older problems, like the computation of a common function between multiple parties, where every party injects its own data into the system, calculates an intermediate result and forwards the resulting machine state to the next participant.
The environment presented in this paper is still limited in performance which makes it currently suitable for small problem sizes and concept studies. These results put some quantitate context on the current limitations of homomorphic program execution and gives a clearer idea of the a scale of problems which can be solved already. Undoubtably performance optimization is one of the main areas of future work. By extending the capabilities of our solution to bilaterally 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. It is possible to inject encrypted data into the encrypted environment, which is sufficient to receive process data from outside the cipher-space. However, the performance benefits this can bring will also induces some problems, like the correctness and consistency of the encrypted code and data.

We are currently determining the requirements for and limitations of a hardware implementation of the encrypted machine. The main challenges will be the memory access with super-wide buses (transferring

[2]available on the web: http://www.hcrypt.com

the encrypted addresses and data) and the hardware implementation of the recrypt procedure.

VI. Summary

In this paper we presented the implementation and performance evaluation of a method to perform the execution of arbitrary encrypted non-linear 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 encrypted data after having been transmitted to the executing host. We have described a method to represent circuits by means of homomorphically encrypted arithmetics. Applying the basic logic function representations, we showed, how to build different microprocessor primitives like memory-access logic and arithmetic operations which can operate in the cipher-space. We then developed a simple CPU- and system-model and presented the reference implementation of our model on top of our implementation of the Smart-Vercauteren encryption scheme. An analysis determined the relationship between our system model and the underlying encryption scheme. We provided performance figures for different key sizes and showed that while homomorphic cryptography will not be saving Cloud computing any time soon, a system such as presented in this paper is suitable to act as a sound basis for further empirical investigation of applied homomorphic encryption.

References

[1] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, LNCS. 2011.

[2] Michael Brenner, Jan Wiebelitz, Gabriele von Voigt, and Matthew Smith. Secret program execution in the cloud applying homomorphic encryption. In *Proceedings of the 5th IEEE International Conference on Digital Ecosystems*, DEST'11. IEEE, 2011.

[3] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology - CRYPTO 2011*, LNCS. 2011.

[4] Ivan Damgård, Sigurd Meldgaard, and Jesper Nielsen. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography*, LNCS. 2011.

[5] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09. ACM, 2009.

[6] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable yao circuits. In *Advances in Cryptology - CRYPTO 2010*, LNCS. 2010.

[7] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, New York, NY, USA, 1987. ACM.

[8] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43:431–473, May 1996.

[9] Michael Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Automata, Languages and Programming*, LNCS. 2011.

[10] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. How to combine homomorphic encryption and garbled circuits improved circuits and computing the minimum distance efficiently.

[11] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security*, LNCS. 2009.

[12] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - a secure two-party computation system. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04. USENIX Association, 2004.

[13] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, New York, NY, USA, 2011. ACM.

[14] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Advances in Cryptology - CRYPTO 2010*, LNCS. 2010.

[15] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography, PKC 2010*, volume 6056 of *LNCS*, pages 420–443. Springer Berlin / Heidelberg, 2010.

[16] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In *Advances in Cryptology - ASIACRYPT 2010*, LNCS. 2010.

[17] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT 2010*, LNCS. 2010.

[18] Andrew C.C. Yao. Protocols for secure computations. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 1982.