A SMART-GENTRY BASED SOFTWARE SYSTEM FOR SECRET PROGRAM EXECUTION

Michael Brenner, Jan Wiebelitz, Gabriele von Voigt and Matthew Smith
Research Center L3S, Gottfried Wilhelm Leibniz Universitaet Hannover, Appelstrasse 9a, Hannover, Germany

Keywords: Homomorphic encryption, Secret program execution, Secure function evaluation, Encrypted processor.

Abstract: Currently generic executable programs can only be encrypted during transmission and storage. To execute the program itself and the data it operates on must be decrypted. If the execution system is not trusted or compromised, both the program code and data are endangered. Recent advances in homomorphic cryptography show how additions and multiplications can be executed in encrypted space, i.e. without decrypting the information, the arithmetic operations themselves are not encrypted. To date, a universal implementation of a homomorphic system, capable of executing arbitrary programs and allowing for practical experiences is still missing. In this paper we present the first method to compute a non-linear arbitrary secret program on an untrusted resource using fully homomorphic encrypted circuits. We use our own implementation of the Smart-Gentry crypto-system as a foundation and define a processor architecture which is capable of executing encrypted programs on encrypted data. Unlike other approaches, such as static one-pass boolean circuit simulations, our system supports read and write memory access, 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. The system represents a first step to show the capabilities of homomorphic encryption in software and system architecture.

1 INTRODUCTION

Fully homomorphic encryption has often been called the *cryptographer's holy grail*. Once the mathematical foundation has been established (Gentry, 2009), we need further procedures and architectures that enable a reasonable application of the encrypted additions and multiplications on single bits. It's essentially this, what the latest homomorphic systems provide. To fully harness the potential of homomorphic encryption, a generic runtime container that is able of executing arbitrary encrypted programs on encrypted data is needed.

In this paper, we present a method to compute such encrypted programs on an untrusted resource using fully homomorphic encrypted circuit representations. We define a sample processor architecture for which we provide a software implementation and present performance figures based on our implementation[1] of the underlying Smart et al. cryptosystem (Smart and Vercauteren, 2010). 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* (Yao, 1986) and different derivates such as (Malkhi et al., 2004) and (Kolesnikov et al., 2009), our system supports non-linear programs, dynamic parameters and subsequent provision of encrypted input data that can easily be written to the encrypted memory. We support programs that render dynamic branch-decisions at runtime, even allow self-modifying code that cannot be represented in a one-pass boolean circuit. The solved problem is different from the classic *multiparty secure function evaluation*, where two parties compute a common function, each delivering a secret portion of the input data that is hidden from the other party (Abadi and Feigenbaum, 1990). Our concept achieves *obliviousness*, as defined by Goldreich (Goldreich and Ostrovsky, 1996) as an implication of sequential circuit simulation. We also solve 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

[1] The implementation can be downloaded from our website at http://www.dcsec.uni-hannover.de/brenner.html

Brenner M., Wiebelitz J., von Voigt G. and Smith M..
A SMART-GENTRY BASED SOFTWARE SYSTEM FOR SECRET PROGRAM EXECUTION.
DOI: 10.5220/0003445802380244
In *Proceedings of the International Conference on Security and Cryptography* (SECRYPT-2011), pages 238-244
ISBN: 978-989-8425-71-3
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

and other interesting approaches are discussed in Section 2. Section 3 introduces the approach of encrypting circuits using homomorphically encrypted bit representations. We also introduce a sample CPU model that is described in detail, both in boolean logic and encryptable arithmetics. We discuss our software implementation and provide basic performance figures in Section 4. Future work and lessons learned from the implementation are presented in Section 5. In Section 6 we give a short summary of our contributions.

2 RELATED WORK

Many different approaches exist, that address security to code execution on remote, untrusted resources. One group of methods is the construction of boolean circuits with encrypted function tables as a function representation. The circuit is then encrypted in some way, to conceal the original function. The contributions of Yao (Yao, 1986), Abadi (Abadi and Feigenbaum, 1990) and Malkhi (Malkhi et al., 2004) are examples for this class of approaches. Their goal is to establish a protocol and an execution container that allows the *Secure Function Evaluation* of a common function, having a secret input from every participating entity. Goldreich and Ostrovsky describe an approach that reduces software protection to online simulation of a program in an oblivious random access machine (Goldreich and Ostrovsky, 1996). Pinkas and Reinman improve the oblivious RAM approach by reducing the protocol complexity (Pinkas and Reinman, 2010).

Another class of concepts addresses security by evaluating encrypted functions and/or data, where most methods apply some sort of homomorphic encryption. Sander and Tschudin (Sander and Tschudin, 1998a) (Sander and Tschudin, 1998b) 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 (Lee et al., 2001). US Patent No. 7,296,163 B2, *Systems and methods for encrypted execution of computer programs* (Cybenko, 2007), tries to solve the problem of encrypted execution by representing boolean XOR and NOT functions as matrix operations and by distributing the calculation over a number of hosts, the results of which have to be merged by a *control computer*.

None of the approaches allows for fully encrypted execution of arbitrary programs on encrypted data with support for read and write memory access, dynamic parameters and non-linear programs that render branch-decisions at runtime.

3 CONSTRUCTION OF AN ENCRYPTED CPU

This section describes basic circuit encryption and different processor primitives that are necessary to construct a CPU and to model random access memory as suggested in (Brenner et al., 2011). We will then transform these fundamental components from the switching functions into the encryptable form.

3.1 Basic Circuit Encryption

Our concept is based on the encryption of circuits in their arithmetic representation. To achieve this, we apply a homomorphic scheme that is capable of multiplication and addition of encrypted bits[2]. To explain the relationship between the switching function and the arithmetic representation we identify the algebraic operations *addition* and *multiplication* with the boolean operations *exclusive disjunction (XOR)* and *conjunction (AND)*, which are sufficient to build arbitrary circuits. The characteristics of binary addition equal the XOR-operation, whereas binary multiplication equals the AND-operation[3]. This allows us to simulate chains of boolean operations by means of simple binary or integer arithmetics. This can be achieved by replacing XOR operators by addition and AND operators by multiplication.

Definition 3.1.1. In boolean expressions the operator $\oplus$ denotes the XOR operation.

Example 3.1.1. The boolean term $r = (a \oplus b) \oplus (a \wedge b)$, which results in a boolean OR-operation can be expressed in integer arithmetics as $r = (a+b)+(a*b)$, assuming a and b being representations of bit values. □

Definition 3.1.2. We use the notation $\circ$ for the composite OR-operation in arithmetics which is defined as $a \circ b = (a+b)+(a*b)$.

[2] An integer based encryption scheme can also be applied. In that case the bit is encoded in a ciphers property of having an even or odd remainder modulo a secret prime key.

[3] The function tables also apply for integer parities: even and odd parities are equivalent to 0s and 1s

3.2 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 over the m-wires, which is a notation that closely relates to a software simulation. The output bit of this single bit memory-column with two address lines can be calculated as

$$c = (\neg a_0 \wedge \neg a_1 \wedge m_0) \vee (a_0 \wedge \neg a_1 \wedge m_1) \vee \\ (\neg a_0 \wedge a_1 \wedge m_2) \vee (a_0 \wedge a_1 \wedge m_3).$$

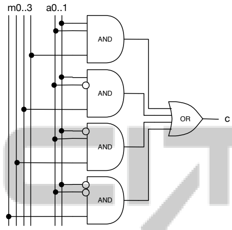

Figure 1: Basic memory circuit.

By extending this function to the required number of address lines and memory columns, we are able to model a memory-circuit of any size. Now we transform the boolean gate logic for this memory circuit into arithmetic, which results in the following expression (see Example 3.1.1 and Definition 3.1.2): $row_0 = ((a_0+1)*(a_1+1)*m_0), row_1 = (a_0*(a_1+1)*m_1), row_2 = ((a_0+1)*a_1*m_2), row_3 = (a_0*a_1*m_3); c = row_0 \circ row_1 \circ row_2 \circ row_3$

Example 3.1. Given the random bit sequence $\{1,0,1,0\}$ as values of a memory column, the sequence $\{0,1\}$ represents the decimal address 2 in binary form. Then the memory access described in arithmetics can be calculated as follows: $row_0 = ((0+1)*(1+1)*1) = 0$, $row_1 = (0*(1+1)*0) = 0$, $row_2 = ((0+1)*1*1) = 1$, $row_3 = (0*1*0) = 0$, $r = 0 \circ 0 \circ 1 \circ 0 = 1$ □

It's important to note that the same result can be achieved when using homomorphically encrypted representations of the bits. So we are able to access encrypted memory providing an encrypted address to the circuit, so the access procedure reveals neither memory address, nor memory content. Assuming that the probabilistic Smart crypto-system provides different cipher representations for a single plain-text bit, 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 wether a particular row holds an encrypted 1 and therefore the calculation of the following row results can be omitted. This provides *obliviousness* because any two memory accesses cannot be distinguished and even the data direction can be kept secret.

To assign a new value to a memory cell, this new value representation is passed as i (*input*) to the access function. For each memory row we generate the new cell value as $m_{new} = (row \wedge i) \vee (\neg row \wedge m)$ with *row* being the row select signal as shown above. This assigns the new bit value i, if the row is selected (and thus has the value 1) and the old value m otherwise. 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: $\forall x \in A$: $m_x = (row_x \wedge write \wedge i) \vee$, $(row_x \wedge \neg write \wedge m_x) \vee$, $(\neg row_x \wedge m_x)$, $c = \bigvee(row_x \wedge m_x)$

Theorem 3.1 (informal statement). Since the result of the access function is a logic combination of all memory cells, the complexity of memory access, which depends of the circuit size ϕ, is an almost linear function. The number of boolean gates $B_{\{1,2,3\}} = \{\neg, \wedge, \vee\}$ that have to be processed in order to determine r during a read access is given as $f(\phi) = (\phi * B_1) + (2 * \phi * B_2) + ((\phi - 1) * B_3)$. □

3.3 Encrypted Arithmetic-logical Unit

To model an encrypted ALU, we apply a similar technique, as we use to implement memory access. Figure 2 shows a simple 1-bit ALU which is capable of an addition and simple boolean operations.

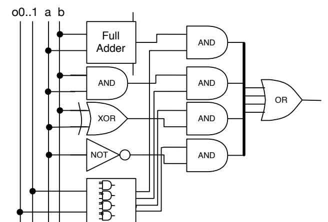

Figure 2: 1-bit ALU circuit.

The ALU essentially consists of a couple of sim-

ple circuits that are applied to the input signals a and b and, in every cycle, produce the operation-specific output. The command-switch $o_0..o_1$, which contains 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. The following series of equations model the ALU in boolean logic, assuming the command-encoding of $\{o_0, o_1\}$ as $\{0,0\} \widehat{=} add, \{1,0\} \widehat{=} and, \{0,1\} \widehat{=} xor, \{1,1\} \widehat{=} not$: $c_{add} = fulladder(a,b)$, $c_{and} = a \wedge b$, $c_{xor} = a \oplus b$, $c_{not} = \neg a$

The following term renders the result of the particular operation, denoted by o_0, o_1: $c = (c_{add} \wedge (\neg o_0 \wedge \neg o_1)) \vee, (c_{and} \wedge (o_0 \wedge \neg o_1)) \vee, (c_{xor} \wedge (\neg o_0 \wedge o_1)) \vee, (c_{not} \wedge (o_0 \wedge o_1))$
Because the ALU function selection and memory access are comparable, we are not going to give further details on ALU circuit modeling. The transformation into integer arithmetic can also be directly derived from the memory model.

To integrate the ALU model in the architecture, we need to determine the size of a *machine word*. In the absence of a physical *data-bus*, this is the common number of memory columns and coupled 1-bit ALUs. The dimension of a memory word can be simply implemented by arranging multiple memory columns in parallel. Our ALU implementation also handles two flags. The *zero*-flag indicates an operation result of zero, or a comparison that yielded equality (since comparisons are often implemented as implicit subtractions, the zero flag usually applies for both comparisons and arithmetic results). The *carry* flag indicates a carry between the 1-bit ALUs.

3.4 Encrypted Branching

As an introduction to branching in the encrypted space, we present the schematic of our CPU- and system-model, because basic data- and program-flow is a prerequisite to understanding branching.

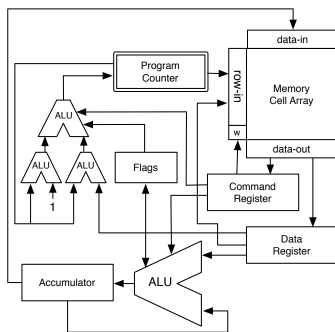

Figure 3: CPU schematic.

Figure 3 shows the basic *data path* in our simple CPU-model. We define a single-cycle, accumulator-driven architecture, which means that every operation is performed in a phased single cycle and that there is only one general-purpose register. A sound introduction to CPU- and system-design is given in (Hennessy and Patterson, 2006).

A good starting point for a brief description is the *Program Counter* that holds the memory address, where the program starts, as an initial value. After accessing memory, the content is stored in the *Command* and *Data Registers*. Arithmetic and logical operations are performed by the main *ALU*, which takes the data register and the *Accumulator* as input and stores the output, according to the command register, back into the accumulator. In case of a load or store operation, the data register acts as a memory address and, when loaded, is overwritten with the addressed memory cell's content. The target register of a branching operation is the program counter. We have unconditional jumps and branches that depend on the system's state, represented by the flag configuration. A 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 flag, like the common *branch-if-zero (bz)* or *branch-if-carry-clear (bcc)*. The triple-ALU in figure 3 handles the entire program flow. It adds a static 1 to the program counter in linear program sequences and adds the data register's content in case of a branch. This branch-logic performs *all* possible branch-address calculations and selects the appropriate address for the program counter. This selection is controlled by the command register and the flag states. Let F be the set of flags, PC the program counter register, DR the data register and CR the command register. The functions $jmp(CR)$, $bcc(CR)$ and $bz(CR)$ take the command register as input and return *true* (a bit representation with odd parity) for the particular command. The next address to be assigned to the PC, following the program flow, is then $\forall x : x \in \{0..wordsize-1\}, PC_x = (jmp(CR) \wedge DR_x) \vee (bcc(CR) \wedge DR_x \wedge \neg F_{carry}) \vee (bz(CR) \wedge DR_x \wedge F_{zero}) \vee (\neg jmp(CR) \wedge \neg bcc(CR) \wedge \neg bz(CR) \wedge (PC+1)_x)$.

3.5 Plugging It Together

Having defined the basic components of a CPU, we are now able to combine these to construct a processor with memory access. The CPU schematic in Figure 3 shows how the components along the data

path have to be connected. We have registers that consist of encrypted bit columns, an ALU that is required for arithmetic and logic operations (the larger main ALU) and a group of smaller ALUs implementing only comparisons and additions to handle the program flow. The memory array consists of the memory cells and the access logic as described above.

The processor cycle is implemented as a phased single cycle comprising four steps:

- FETCH1 read memory cell pointed at by program counter
- FETCH2 read memory cell pointed at by fetched operand
- EXEC execute operation in command register
- WRITE write result or refresh old memory value

Every single cycle has to perform three memory access operations. This is required to achieve obliviousness to make any two processor cycles indistinguishable.

4 IMPLEMENTATION & PERFORMANCE

We provide prototype implementations of our execution engine concept in Java and C. This section describes the basic system properties and performance figures for selected components of the C implementation. Figure 4 shows the layered architecture of the runtime environment.

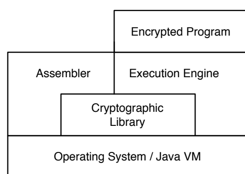

Figure 4: System Architecture.

The prototype implementation of the processor outlined in Section 3 use 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 allows for a simple processor architecture with a single fetch cycle for opcode and operand. If a memory location is written to, only the data compartment is modified, whereas the command compartment remains untouched. The actual processor implementation, which is independent of the underlying crypto-system, executes 129,397 boolean gates for one cycle including oblivious memory access on 256 13-bit words. The entire processor circuit consists of 44,980 XOR and 15,476 NOT gates and 68,933 AND gates. An unencrypted cycle (i.e. the processor implementation is running with deactivated crypto-library) takes between 2 and 3 milliseconds on our test configuration including memory access and all CPU logic. Our test setting consists of a 2.4 GHz Intel Core 2 Duo platform with 4 GB of 667 MHz DDR2 SDRAM. The processor prototype is executed on a virtual 1-CPU machine (VirtualBox) running a 32-bit Linux with 2.6 kernel and 1 GB of RAM.

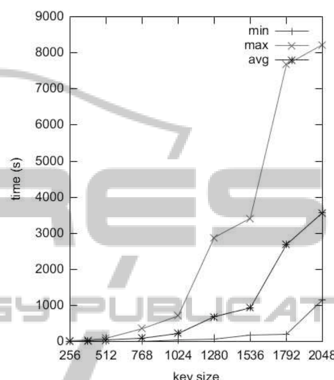

Figure 5: Key Construction.

We have implemented the Smart-Gentry cryptosystem as a cryptographic library for our system. It is configured in a couple of small to medium settings regarding the sizes of key coefficients and ciphers. The key coefficients are a set of integers and contain the components for the decryption hint of the Smart-Gentry public key. Our implementation applies a set of eight components in the public key and serves as a proof-of-concept configuration. Figure 5 shows the time consumption of key generation for different key sizes. The key size parameter describes the integer range of a key coefficient as $\pm 2^{x-1} \in \mathbb{N}$. Key coefficients and the actual ciphers (the encrypted bit representations) have the same size. Key generation times range from an average of 3 seconds for a key size of 256 to an average of half an hour for size 2048. To track the noise of the ciphers, we attach a numeric noise counter to every cipher. We assume that an addition (XOR and NOT gates) increases the noise by 1 while charging the multiplication (AND gates) with a value of 1000. The crypto-library triggers a recryption of a cipher when exceeding a noise measure of 2000.

Figure 6 depicts the access time for different key sizes and is measured as seconds per row of 13 bits (a memory word). Since the access duration grows depending on the memory size, compact programs and data are the key to tolerable runtimes. The sizes of key

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

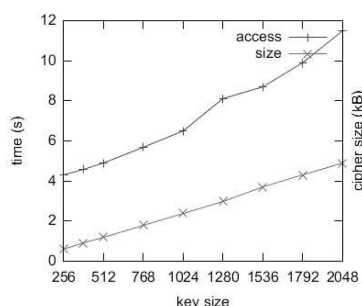

Figure 6: Memory Access.

coefficients and ciphers are also shown in that figure.

5 OPEN ISSUES & FUTURE WORK

The environment presented in this paper has proof-of-concept capabilities and a dependency between memory size and performance, which makes it 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. It is possible to inject encrypted 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. A possible field of application is *Cloud Computing*, where small- and medium-scale compute jobs are performed which have high privacy requirements. The establishment of an appropriate system- and application-architecture will be the key to integrate our concept into existing cloud applications and environments, to face new security requirements of mobile code and distributed applications.

6 SUMMARY

In this paper we presented the first method to perform the execution of arbitrary encrypted programs, operating on encrypted data. In contrast to other solutions, the code as well as the processed data, held entirely in the cipher-space, still remain dynamic and can be provided with data after having been transmitted to the executing host. We described a method to represent circuits by means of homomorphically encrypted 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 on top of the Smart-Gentry 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 our system is suitable to act as a sound basis for further empirical investigation of applied homomorphic encryption.

REFERENCES

Abadi, M. and Feigenbaum, J. (1990). Secure circuit evaluation. *Journal of Cryptology*, 2:1–12. 10.1007/BF02252866.

Brenner, M., Wiebelitz, J., von Voigt, G., and Smith, M. (2011). Secret program execution in the cloud applying homomorphic encryption. In *Proceedings of the 5th IEEE International Conference on Digital Ecosystems (DEST 2011), to appear*, DEST'11, USA. IEEE.

Cybenko, G. (2007). System and methods for encrypted execution of computer programs.

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, New York, NY, USA. ACM.

Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious rams. *J. ACM*, 43:431–473.

Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture. A Quantitative Approach.* Academic Press.

Kolesnikov, V., Sadeghi, A.-R., and Schneider, T. (2009). How to combine homomorphic encryption and garbled circuits - improved circuits and computing the minimum distance efficiently. In *Signal Processing in the Encrypted Domain*, SPEED'09, Lausanne, Switzerland. SPEED Project.

Lee, H., Alves-Foss, J., and Harrison, S. (2001). Securing mobile agents through evaluation of encrypted functions. Technical report, Center for Secure and Dependable Software Computer Science Department, University of Idaho.

Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. (2004). Fairplay - a secure two-party computation system. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, Berkeley, CA, USA. USENIX Association.

Pinkas, B. and Reinman, T. (2010). Oblivious ram revisited. In Rabin, T., editor, *Advances in Cryptology CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519. Springer Berlin / Heidelberg. 10.1007/978-3-642-14623-7_27.

Sander, T. and Tschudin, C. (1998a). Protecting mobile agents against malicious hosts. In Vigna, G., editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 44–60. Springer Berlin / Heidelberg. 10.1007/3-540-68671-1_4.

Sander, T. and Tschudin, C. F. (1998b). Towards mobile cryptography. *Security and Privacy, IEEE Symposium on*, 0:0215.

Smart, N. and Vercauteren, F. (2010). 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.

Yao, A. C.-C. (1986). How to generate and exchange secrets. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:162–167.