# POSTER: Caching Oblivious Memory Access

## An Extension to the HCRYPT Virtual Machine

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

## ABSTRACT

Efficient homomorphic encryption enables the construction of an encrypted computer system. Previous work has shown how this can be achieved using only arithmetic representations of simple demultiplexer circuits. This poster extends the results by introducing a caching mechanism for oblivious memory access, by far the most time-consuming building block of a recently proposed sample machine architecture. The construction allows to significantly accelerate homomorphically encrypted machine operation while still preserving obliviousness of memory access, control unit operation and functional components.

## Categories and Subject Descriptors

E.3 [**Data Encryption**]: Public Key Cryptosystems

## Keywords

homomorphic encryption, implementation, oblivious memory access

## 1. INTRODUCTION

Gentry's fully homomorphic encryption scheme seemed to be the saviour of confidential delegation to remote resources - but only for a second. Regardless of its elegance, it proved slow when applied in a scenario where large circuits are encrypted. Nevertheless, homomorphic encryption is one of the hottest topics in current research. The other direction of research that aims at making encrypted computing feasible is the construction of circuits and protocols that arrange the underlying cryptographic primitives in an efficient way. This work belongs to the latter.

In previous publications we showed how to construct an encrypted universal machine using only very basic circuits, in particular demultiplexers or selectors [1]. We then proved the concept with an implementation of a simple machine model [2] (the *hcrypt* machine) and evaluated the performance of such a design using our implementation [3] of the homomorphic Smart-Vercauteren scheme [4]. The evaluation clearly showed that the memory access is by far the most time-consuming operation when implemented with encrypted circuits.


*CCS'13,* November 4–8, 2013, Berlin, Germany.
ACM 978-1-4503-2477-9/13/11.
http://dx.doi.org/10.1145/2508859.2512501.

## 2. THE MACHINE MODEL

The sample machine that we base our investigation on is the *hcrypt* virtual machine that can be downloaded from our website[1]. We use a simple accumulator-based scheme with a Harvard-style memory architecture. The sample machine features a small memory array of 256*13 bits, resulting in a ratio of roughly 99:1, comparing the number of gates forming memory and processor circuits. Obviously, in settings with reasonable memory sizes, the execution time of the processor gates is negligible. By executing the arithmetic representations of the gates under encryption, the resulting machine can securely execute any code on any data without the need for decryption. This model allows branching programs, loops and even self-modifying code.

### 2.1 Oblivious Memory Access

Our original approach uses the following pattern to obliviously calculate the result of a memory access: The result $r$ of the switching function for a 4-bit memory column $m$ is given as $r = (\neg a_0 \wedge \neg a_1 \wedge m_0) \oplus (a_0 \wedge \neg a_1 \wedge m_1) \oplus (\neg a_0 \wedge a_1 \wedge m_2) \oplus (a_0 \wedge a_1 \wedge m_3)$ with $a_x$ being the address selector. This can easily be transformed into an arithmetic representation $r' = ((a'_0 + 1') * (a'_1 + 1') * m'_0) + (a'_0 * (a'_1 + 1') * m'_1) + ((a'_0 + 1') * a'_1 * m'_2) + (a'_0 * a'_1 * m'_3)$ with $x'$ denoting the encrypted value of a bit $x$ under a fully homomorphic encryption scheme. This calculates the encrypted memory value for an encrypted address over an encrypted memory array.

To ensure obliviousness, every single memory access is required to calculate the switching function translated into Line 8 in function `Ma()` for the entire memory array of size $ROWS * WORDSIZE$. We iterate over all addresses $i$ in the address space $\{0, ROWS\}$ and every bit $j$ of a memory word. The access logic is depicted in Figure 1.

```
1: res Ma(adr)
2: {
3:   for(i = 0..ROWS)
4:   {
5:     sel = RowSelect(i,adr);
6:     for(j = 0..WORDSIZE)
7:     {
8:       res.j  = (A[i].j * sel)+(res.j  * !sel);
```

[1]https://hcrypt.com

```
 9:     }
10:   }
11:   return res;
12: }
```

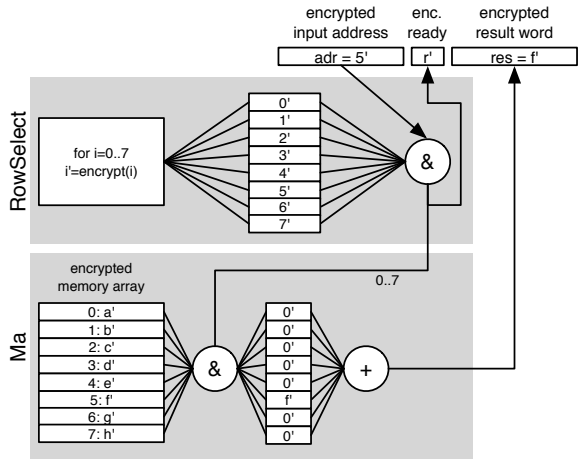


**Figure 1: Memory access schematic**

During memory access, the function `RowSelect()` computes the equality of the address iterator $i$ and the memory address $adr$ to be accessed, resulting in an encrypted 1 on equality and an encrypted 0 otherwise (because all bit representations are encrypted).

```
 1: sel RowSelect(i,adr)
 2: {
 3:   sel=Encrypt(1);
 4:   for(j = 0..log2(ROWS))
 5:   {
 6:     temp = i.j + adr.j + Encrypt(1); //a XOR b XOR 1 => (a==b)
 7:     sel=sel*temp;
 8:   }
 9:   return sel;
10: }
```

The function `RowSelect()` is called iteratively by `Ma()`. The row result of each iteration is added to the intermediate result *res*. As a consequence, the result remains stable, as soon as the addressed memory row was found and the memory content was added to the intermediate result (which up to this point is 0). Any subsequent iteration can only add zeros, as the address comparison can't produce another positive. Storing the comparison result in an indicator flag $r'$, it's possible to use the result *res* before the remaining memory words have been processed.

# 3. OBLIVIOUS CACHING

Computing the address equality in fact implements an associative item selection usually found in caching technology. This is the basis to extend the existing machine architecture. We assume that the single-threaded execution pattern is replaced by (at least) three parallel threads executing two memory blocks and the processor control unit circuits. The memory blocks are extended by a number of encrypted cache words which hold recently accessed memory words along with the corresponding memory addresses (tags) as depicted in Figure 2. The memory access function starts by associatively matching the currently accessed address with the cached tags. In case of a cache hit the corresponding word is copied from the cache to the output register. In either case, cache hit or not, the boolean address matching result is also stored in the flag $r'$, possibly indicating the availability of the requested memory item.

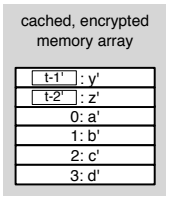


**Figure 2: Cached Memory Block**

## 3.1 An Old Acquaintance: The Wait State

Splitting memory into multiple parallel instances requires a synchronization step to collect the results from the memory banks. This can easily be implemented by putting the memory management circuit (and as a dependency the control unit) into wait mode. To keep the control unit oblivious to an observer, control unit operation must not be halted but any operation in the *hcrypt* state machine may only be executed if memory is in sync, which is the case, as soon as the first memory bank reports a hit. To practically implement the state machine wait mode, the write-back phase is extended by an additional conjunction which effectively skips the modification of the machine state unless the sync flag $s'$ is 1. As an example, the write-back circuit for the program counter which reads

$\forall x : x \in \{0..7\}$,
$PC_x = (jmp(CR) \wedge DR_x) \vee (bcc(CR) \wedge DR_x \wedge \neg F_{carry}) \vee (bz(CR) \wedge DR_x \wedge F_{zero}) \vee (\neg jmp(CR) \wedge \neg bcc(CR) \wedge \neg bz(CR) \wedge (PC+1)_x)$.

for the 8-bit program counter $PC$, data and command registers $DR$, $CR$, a flag register $F$ and selectors that return boolean values that indicate one of the opcodes $\{jmp, bcc, bz\}$ in $CR$, would be extended to

$PC_x = \underline{PC_x \vee s' \wedge}((jmp(CR) \wedge DR_x) \vee ... \wedge (PC+1)_x))$.

where $s'$ is the sum over all ready flags $r'$. Extending this approach to all machine state variables allows the control unit circuit to loop in an non-operational state which cannot be detected by an observer.

## 3.2 Caching Strategies

Due to space limitations, we only sketch a first-in-first-out principle. The advantage of this paradigm is that it does not require additional encrypted circuitry to substitute the cache items. The executing entity simply shifts the cache array by one item and writes the encrypted result *res* into the $t-1$ position of the cache array along with the address *adr* as the corresponding tag. The caching mechanism shown here can apply any caching strategy. Advanced mechanisms like a least-recently-used or counter method would need encrypted counters and comparators to calculate the substitution candidate. The following subsections sketch further measures that can optimize access to previously uncached memory items.

## 3.3 Memory Bank Interleaving

A trivial *hcrypt* looping program shows an obvious problem with a sequential order distribution of the address space over the memory banks.

```
1: count   .integer 20  // 20 iterations
2: start   La count     // counter -> accumulator
3: loop    SEC          // set carry
4:         SUB 1        // decrease by 1
5:         BNE loop     // repeat if >0
6:         END
```

Assuming that the size of a memory bank is larger than the loop, the memory fetch cycles block each other because the memory items holding the program code are all stored in the same bank and obliviousness demands that each memory access operation iterates over the entire bank. To avoid this blocking behaviour, the address space has to be distributed over the available memory banks in an interleaved manner as shown in Figure 3. The example also shows that the number of parallel memory banks and the interleave factor have an significant impact on efficiency: Regarding the cycle phases of the sample *hcrypt* machine architecture, it is even necessary to arrange the memory in a way such that a command word and a target address in absolute address mode (a reference by memory address, like in the sample program) are stored in separate banks. However, this is not an issue specific to the *hcrypt* machine but to any phased cycle processor architecture that offers absolute addressing.

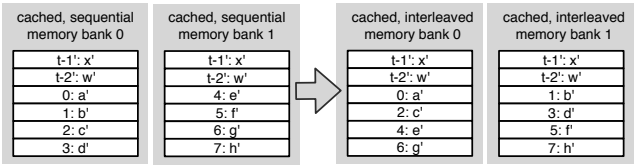


**Figure 3: Interleaved Memory**

To optimize the memory layout according to a particular program, the entire memory block can be arranged cache-style as depicted above. The `RowSelect()` function is then replaced by pre-computed (i.e. pre-encrypted) addresses for all memory words. This allows to store the items of a memory bank in arbitrary order but sacrifices space for flexibility.

## 3.4 Deferred Memory Bank Evaluation

Consider the sample program being stored in two uncached, parallel memory banks with interleave 2 as shown in Figure 3 but with 8 items each. Assume that the evaluation of each item takes a uniform timeslice $t = 1$. The control unit issues the fetch operations for the command word and the argument at two sequent memory-idle timeslices. The schedule for the sample program is shown in Figure 4.

The chart shows that the undeferred schedule for the first operation (`La count`) causes 7 wait states (w) between the two fetch phases (f) of one control unit cycle. This results in 16 busy (-) timeslices to load the first single operation from memory, even though bank 1 generates the ready signal (X) after just one timeslice. This, by the way, is the behaviour of a sequential execution without parallel memory banks, which clearly shows, that the naïve approach of simply executing multiple instances of memory evaluation in parallel

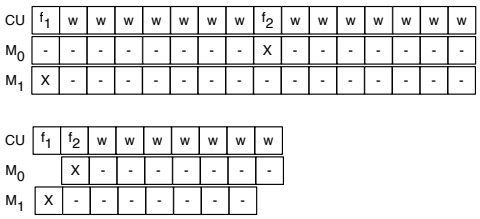


**Figure 4: Deferred Evaluation**

has absolutely no effect. Slightly deferring the evaluation of bank 0 by just one timeslice optimizes the program flow, such that the operation now takes only 9 timeslices.

The generation of the ideal memory layout is a machine- and program-specific compile-time issue and will not be covered here. The executing entity which computes on the encrypted circuits only needs to know the access mode or the interleave factor in order to modify the iteration steps in the `Ma()` function. Note that this, to some degree, reveals information about the program structure and therefore conflicts with a strong notion of obliviousness.

## 3.5 Write-Back

To assure memory consistency, write access is required to process the cache entries first. If the cache contains a target address tag then the corresponding entry is updated, which immediately leads to a valid memory state, as all subsequent read operations also query the cache first.

# 4. CONCLUSION

This work presents a mechanism for caching oblivious memory access implemented with circuits under a fully homomorphic encryption scheme. The construction applies parallel evaluation of different memory banks and offers solutions to various subproblems like access synchronization, prevention of access blocking by interleaving and optimization by deferred evaluation in a scenario that demands obliviousness. We showed the concept as an extension to the encrypted *hcrypt* machine.

To optimize the program flow and the program-specific memory layout, further consideration of established methods in compiler-based operation scheduling is recommended.

# 5. REFERENCES


[1] M. Brenner, J. Wiebelitz, G. v. Voigt and M. Smith: *A Smart-Gentry based Software System for Secret Program Execution.* 6th International Conference on Security and Cryptography (SECRYPT 2011), SciTePress, 2011

[2] M. Brenner, H. Perl and M. Smith: *How Practical is Homomorphically Encrypted Program Execution? An Implementation and Performance Evaluation.* 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 2012

[3] H. Perl, M. Brenner and M. Smith, *Poster: An Implementation of the Fully Homomorphic Smart-Vercauteren Cryptosystem*, ACM CCS'11, Proceedings of the 18th ACM conference on Computer and communications security 2011

[4] N. P. Smart and F. Vercauteren, *Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes*, Public Key Cryptography - PKC 2010, Springer 2010