On the Security of Practical and Complete Homomorphic Encrypted Computation

Peter T. Breuer[1] and Jonathan P. Bowen[2]

[1] Hecusys LLC, Atlanta, GA, USA; e-mail: ptb@hecusys.com
[2] London South Bank University, London, UK; e-mail: jonathan.bowen@lsbu.ac.uk

Abstract. Security with respect to the operator as an adversary is considered for processors supporting unbounded general purpose homomorphic encrypted computation. An efficient machine code architecture is defined for those platforms and it is proved that user programs expressed in it are cryptographically obfuscated, guaranteeing privacy though they, their traces and (encrypted) data are visible to the operator. It is proved that encrypted user data cannot be deciphered by the operator, nor may programs be altered to give an intended result. A compiler is defined and it is proved that any recompilation produces uniformly distributed random variations in runtime data, supporting cryptographic obfuscation.

1 Introduction

Fully *homomorphic encryption* (FHE) as proposed by Gentry in 2009 [18] and subsequent improvements [14, 45, 5] and even new directions [20] is approaching a point where it might begin to find practical applications, though it remains for the moment an extremely slow technology even when implemented in hardware [19] (of the order of 1 bit-op per second on vector mainframes). Slow or not at present, the forthcoming applications encompass 'multiparty' calculations, in which several parties jointly work on common but heterogeneously encrypted data in the Cloud, thanks to protocols such as those developed by Lopez-Alt et al. [35], under which users and operator cannot snoop on one another.

But a conflicting view is asserted in van Dijk et al. [15], where it is shown that a cryptographic solution cannot ensure the requisite multiparty privacy in the Cloud for *arbitrary* computations on the server, whatever the protocol. This paper will disagree, asserting that new processing contexts have fundamentally changed the landscape, enabling just that. It will be shown that van Dijk et al.'s argument in fact establishes privacy, with the right hardware underneath.

In support of that view, some prototype processor architectures such as Ascend [17] by Fletcher et al. (van Dijk is a coauthor), which obscure instructions and data from the operator's view by a variety of means, cryptographic and physical, have been shown by their authors to be secure platforms for private computations in the Cloud. The arguments parallel those in this paper. The argument of van Dijk et al. in [15] shows that privacy of arbitrary computations on a server in the multiparty setting is equivalent to generic program *obfuscation*

[23], and defers to a result of Barak et al. [3] that the latter is impossible. But Barak et al.'s result is based on assumptions of what code looks like and what can be told from it that are invalid for the codes and processors considered here, or the formal results in this paper could not be.

Ascend, for example, makes looking at running code impossible by means of hardware protections and protects it on the way to the processor via encryption. If the code and I/O is encrypted, and the processor executes it in controlled isolation to match defined statistics on observables, as is the case with Ascend, then it *is* obfuscated from the operator. The idea has surfaced many times over the years (e.g., [25, 26]) and success means doing it as well as Ascend does.

If not entirely encrypted, code may still be encrypted in part such that it is impossible to tell anything significant by what is readable. That is the case in the HEROIC architecture [46], which has just one non-control instruction type [47] and the unencrypted part is the same between instructions. A proof in this paper (Section 5, Theorem 1) shows that HEROIC traces, though visible, do not give away information about what the data means (the data circulates in encrypted form in the HEROIC processor), nor may a malfeasant operator change the data to their own choice (Section 5, Theorem 2).

The generalisation of HEROIC's approach is embodied in the KPU (Krypto Processor Unit; in which a modified arithmetic has been substituted in an almost conventional CPU design) [9]. Code is visible to all and data is encrypted, as in HEROIC, but there are potentially as many instructions as in any conventional processor. This paper proposes a machine code *instruction set architecture* for KPUs that leaves their traces formally uninformative too (Section 6, Theorem 3). In summary, Barak et al.'s result seems overtaken by technology. The platforms

1)	Ascend:	encrypted code, I/O and memory, physical protections;
2)	HEROIC:	indistinguishable code, encrypted data, I/O and memory;
3)	KPU:	distinguishable code, encrypted data, I/O and memory

will here be taken as paradigms for general purpose 'homomorphic encrypted computation' (HEC), which term will be refined shortly as security is discussed in Section 2 below, where vulnerabilities associated with computational completeness will be disclosed that are later in this paper closed first for HEROIC, then the KPU, culminating with the definition of an 'obfuscating' compiler for the target, and proof of the obfuscation property (Section 7, Theorem 4).

2 Completeness and Security in HEC

A fully homomorphic encryption $[\,\cdot\,]_{\mathcal{E}}$ is such that adding up or multiplying two encryptions respectively adds or multiplies the unencrypted numbers 'underneath': $[x{+}y]_{\mathcal{E}}{=}[x]_{\mathcal{E}}{+}[y]_{\mathcal{E}}$, $[x\,y]_{\mathcal{E}}{=}[x]_{\mathcal{E}}\,[y]_{\mathcal{E}}$. Sometimes applications require a fixed small number of multiplications and then a *somewhat homomorphic encryption* (SHE) may do. A SHE is without the periodic renormalisations that are a hallmark of known FHEs, resulting in faster and smaller encryptions. Without renormalisation, arithmetic eventually takes numbers out of range, for example nearing $2m$ in a calculation mod m. But before that happens the calculation will have finished. Lauter et al. [32] quote a scheme with block-size about 43.5KB and encrypted 1-bit addition in 1ms, multiplication in 43ms, achieved on a 2.1GHz Intel Core 2 Duo in 1GB of RAM. That roughly equates to a 30Hz (sic) Pentium.

More speed comes with substituting a *partially homomorphic encryption* (PHE), if feasible. That is homomorphic with respect to one of addition and multiplication, not both. El Gamal [16] and Paillier [40] are well-known additively homomorphic schemes. Paillier converts addition on unencrypted numbers to multiplication mod $m{=}n^2$ on encrypted numbers, where $n{=}pq$ is the product of two primes. The value n is public, but factoring is infeasible when it is 1024 bits long or more, when the difficulty of factoring n via best-known sieves (minimal running time $e^{\sqrt{1024\log 1024}} \sim 2^{121.5}$ operations [33]) is about the same as brute-forcing a 128-bit key. An encrypted number mod $m{=}n^2$ is 2048 bits long and encrypted add takes 10us (20K cycles) on the 2.1GHz Intel Duo.

A further change is to not only embed the encrypted arithmetic as hardware, as IBM did in [19], but to base all of the processor around it, using it as its arithmetic logic unit (ALU). The HEROIC processor is built like that. Its core does Paillier 2048-bit encrypted addition in about 4K cycles on a 200MHz programmable hardware platform, for about 20us per encrypted (16-bit) addition, 1.25us per bit, roughly equating to a 25KHz Pentium. HEROIC is an example of a processor that *does all its arithmetic encrypted.*

Such processors ('KPUs') are characterised in [7], which shows that changing the ALU in a processor design to 'encrypted working' is enough to support arbitrary unbounded computation with encrypted data. That works even if the encryption has no homomorphic properties [10], because the system as a whole is homomorphic. Designs have been tested in which AES [13] (the US 'Advanced Encryption Standard') serves as encryption, and computation speeds equal to a 600MHz Pentium are reported in [9]. The difference from a security point of view in AES as against Paillier is that a key must be embedded in the processor in order to configure the arithmetic. That may be done at manufacture, using Smart Card fabrication techniques [29] for protection, or a Diffie-Hellman circuit [11] may safely load the key to the processor in public view, with authentication.

Unbounded homomorphic encrypted computation is supported by HEROIC and the KPU. That is distinct from the special-purpose calculations proposed for FHEs and PHEs. While those include web-based voting [1], public-key encryption schemes [42], and *private information retrieval* (PIR) schemes [30] (encrypted databases and queries) and machine learning [21], as well as image eigenface analysis [31], a generic HEC platform should be capable of computational tasks individually of unbounded complexity, *not* defined beforehand.

Completeness: Even to multiply two encrypted numbers, the Paillier encrypted addition in HEROIC does not suffice. A comparison operation between encrypted numbers is required, then it may be implemented as a subroutine. HEROIC provides a *table of signs* [3] for 16-bit encrypted numbers, saying whether the

[3] Paillier incorporates a random 'blinding factor' into encrypted numbers $[x]_{\mathcal{E}}$: an arbitrary multiplier $r^n \bmod m$, where $n{=}pq$ and $m{=}n^2$ is the public modulus. Decryption involves raising to the power of the order of the multiplicative group mod n, so the

number under the encryption is positive (high bit unset) or negative (high bit set), allowing $x<y$ to be determined from the sign for $[x-y]_{\mathcal{E}}$.

With encrypted addition, comparison, one nonzero encrypted constant, plus iteration or recursion, any computable function can be programmed [41], forming a generic platform for homomorphic encrypted computation. HEROIC is very nearly minimal for what is required for computational completeness and both Ascend and KPU are more comfortable for the programmer because they offer the panoply of conventional machine code instructions as programming interface.

Is HEC secure? Being able to run any computable function is more than was envisaged when calculations with encrypted addition and multiplication were proposed as applications for homomorphic encryptions. To an extent even those are not secure because an attacker can take an encrypted value $[x]_{\mathcal{E}}$ and calculate

$$[x-x]_{\mathcal{E}} = [0]_{\mathcal{E}}$$

using subtraction to obtain an encrypted zero (subtraction may be available either as a machine instruction, or as a subroutine). They might go on to recognise that code for zero in some encrypted data. Repetition should produce as many different encryptions of zero as may be wished for the purposes of comparison.

Even in the absence of an explicit subtraction operation, because FHEs generally implement 1-bit arithmetic, zero can be obtained via encrypted addition:

$$[x+x \mod 2]_{\mathcal{E}} = [0]_{\mathcal{E}}$$

while in HEROIC's 16-bit 2s complement arithmetic under Paillier:

$$[2^{16}x \mod 2^{16}]_{\mathcal{E}} = [0]_{\mathcal{E}}$$

where the scalar multiplication represents repeated addition. Moreover, HEROIC does division in a subroutine, and an attacker may use that subroutine to get an encrypted 1 from any observed $[x]_{\mathcal{E}}$ that happens not to be an encrypted zero:

$$[x/x]_{\mathcal{E}} = [1]_{\mathcal{E}}$$

Statistically, many such $[x]_{\mathcal{E}}$ will be observed at runtime. The code for the division subroutine is a weakness in itself, because one of the constants in it should be an encrypted 1. Guess which it is and, by repeated addition, an attacker may efficiently build an encryption of any number K from its binary representation:

$$[2^{k_1}1 + \cdots + 2^{k_j}1]_{\mathcal{E}} = [K]_{\mathcal{E}}$$

Conversely, given an encrypted 1 and HEROIC's comparison operation, any encrypted number $[x]_{\mathcal{E}}$ can be decrypted by comparing it with each constructed integer $[K]_{\mathcal{E}}$ in turn, or, more efficiently, by deducing its binary digits one by one, comparing with and subtracting $[2^k]_{\mathcal{E}}$ when $2^k \le x < 2^{k+1}$.

So homomorphic encrypted computation has security questions to answer.

---

r^n becomes 1 mod n^2 and does not affect anything. The r^n may be removed using the additional secret that the r have been chosen with multiplicative order k mod n, where k is a certain divisor of the order of the multiplicative group mod n. Then $[x]_{\mathcal{E}}^k$ mod m is an unblinded encryption of kx. If the latter is 1, then, as unblinded Paillier encrypts zero to 1, this is a runtime test for $x=0$, which theoretically, but impractically, can substitute for HEROIC's static table of signs.

3 Functions Computed with a FHE

The common claim that 'any' function can be implemented with a FHE needs clarification. Both addition and multiplication have the property that if all inputs are set to zero, the output is zero, therefore any combination has that property:

Proposition 1. *Any function $f(\boldsymbol{x})$ implemented by an FHE or PHE takes encrypted zero inputs $\boldsymbol{x}{=}\mathbf{0}$ to an encrypted zero output $f(\mathbf{0}) = 0$.*

The proof is the sentence above the statement. Conditionals are admissible too: choice $f(\boldsymbol{x}){=}t?f_1(\boldsymbol{x}){:}f_2(\boldsymbol{x})$ between functions that produce zero when all inputs are zero still produces zero when all inputs are zero, no matter which way t goes.

Therefore 'logical negation', taking 0 to 1, 1 to 0, cannot be implemented via a FHE. Of the 16 boolean operators $f(x, y)$, 8 can and 8 cannot be done using addition and multiplication alone. But the 'cannot' are of the form $f(x, y){=}1{+}p(x, y)$, where p 'can'. So any boolean operator $f(x, y)$ may be implemented by putting $z{=}1$ in $q(x, y, z){=}z{+}p(x, y)$, and addition, multiplication and 1 suffice together.

In conclusion, encrypted calculations generally require a *fixed nonzero constant*, usually 1, comprising an extra secret that needs to be guarded well.

4 Functions in HEC

When k-bit binary arithmetic with $k{>}1$ is implemented under a FHE or PHE, as in HEROIC, or is implemented in a more general system for homomorphic encrypted computation such as a KPU, then there are relatively fewer functions that can be implemented using additions and multiplications alone. Those which 'can' are characterised by Proposition 1 taken also mod 2, 4, ... , 2^{k-1}. Of the standard computer arithmetic functions, aside from logical negation and 'bitwise complement' which are already ruled out by the proposition, right shift ($4{\gg}2{=}1$ is not 0 mod 2) and division ($4/4{=}1$ is not 0 mod 4) are ruled out by this.

So a system for general purpose HEC, even using a FHE or PHE, must embed an encrypted nonzero constant for future use, or demand it as input each time.

5 Secure HEC

Nevertheless, there are ways of running HEC securely. The 'OI' in 'HEROIC' stands for 'one instruction': the machine code has only one significant instruction, an add-and-branch, which sets a value in one memory location and branches if the value in another is low enough. Instructions may be considered compounds of *addition of a constant* $y{\leftarrow}x{+}k$ and branches based on *comparison with a constant* $x{<}K$. Those two, together with control instructions, are sufficient to perform any computation, as evidenced by Conway's well-known 'Fractran' programming language [12], where those are the only high-level instructions.

Consider a program *C written using only those two instructions.* Ensure *no collisions* between encrypted constants (a) $[k]_{\mathcal{E}}$ and (b) $[K]_{\mathcal{E}}$ in the code nor of those with (c) runtime encrypted values, by padding or blinding appropriately:

Theorem 1. *No method of observation exists by which the operator who does not possess the key may decrypt the output $[y]_\mathcal{E}$ of the program C.*

Proof. *Suppose for contradiction that the operator has a method $f(T,C) = y$ of knowing that the output $[y]_\mathcal{E}$ of C encrypts y, having observed the trace T. Now imagine that every number has $h{\neq}0$ added to it under the encryption. The additions $[y{\leftarrow}x{+}k]_\mathcal{E}$ in C still make sense, adding k under the encryption to a number that is h more than it used to be to get a number that is h more than it used to be. Comparisons $[x{<}K]_\mathcal{E}$ in C need changing, however, because the x, which are h more than they used to be, now need to be compared with K' equal to $K{+}h$ for the program to make sense. So the branch instructions in the program must be modified to contain $[K']_\mathcal{E}$ instead of $[K]_\mathcal{E}$. To the operator, the new program code C' 'looks the same', $C' \sim C$, because one encrypted number is as meaningful as another without the key (by the 'no collisions' hypothesis, the operator cannot tell either by a new collision or lack of an old one that the K have changed), and the program trace T' looks the same up to the encrypted numbers in it, which the operator cannot read, so it looks the same, $T' \sim T$, and the method f must declare the output of C to be $f(T',C'){=}f(T,C){=}y$. But it is not $[y]_\mathcal{E}$ but $[y{+}h]_\mathcal{E}$, so the method fails.* $\square$

The heart of the proof is that to the operator one program code and trace can look the same as another that does something different. In the Ascend architecture that is justified because access to the code and the trace is restricted, respectively via encryption and access control. In the HEROIC design that works because enough machine code instructions look alike, up to the encrypted (hence unreadable by the operator) data they contain, to mask the issue. But KPUs run more instruction types and some might comprise a readable program. If the program is just 'sub[tract] r,r,r' (replace $[x]_\mathcal{E}$ in register r with $[x - x]_\mathcal{E}$), for example, then the (encrypted) zero result is certain. The operator might also copy and rearrange other instructions to decipher what is going on, so there is a question over what instructions a KPU should offer, which Section 6 will settle.

An elaboration of the proof of Theorem 1 establishes the result for plural outputs y too. Consider program C again, with the 'no collisions' arrangement:

Theorem 2. *There is no method by which the operator can alter program C using just add and compare with constant instructions to get intended outputs $[y]_\mathcal{E}$.*

Proof. *Suppose for contradiction that the operator builds a new program $C'{=}f(C)$ that returns $[y]_\mathcal{E}$. Then its constants $[k]_\mathcal{E}$ are found in C and its constants $[K]_\mathcal{E}$ likewise, because f has no way of arithmetically combining them (the disjoint subspaces condition means they cannot be combined arithmetically in the processor and the operator does not have the encryption key). Theorem 1 (plural y) says the operator cannot read outputs $[y]_\mathcal{E}$ of C', yet knows what they are.* $\square$

This partially answers whether an attacker can turn the instructions of general HEC to decrypting program data. The answer is 'no', if instructions available are confined to just add a constant/branch on compare with a constant, as in

HEROIC. The answer is also 'no' if the attacker is prevented from running their own programs on user data, as in Ascend. The result will be extended below to apply to generic KPU machines, which may have more and different instructions.

6 FxA Instruction Set

The argument above may be extended to cover two parameter additions, one and two parameter subtractions, multiplications, left shifts, *provided* each instruction is followed by addition of some constant k (which may be zero). That can be enforced in the processor architecture and supported by the compiler, but there is also an opportunity there to co-opt a single *fused multiply and add* (FMA) machine code instruction to cover for all the instructions just mentioned.

That is convenient because FMA is nowadays seen as the ideal unit of parallel computation, having been introduced for that purpose by AMD and Intel in 2011/12/13, with the FMA3/4 instruction sets. Denote by a *fused anything and add* (FxA) instruction set one in which no arithmetic instruction appears except as the fused compound with a trailing (or preceding) constant addend. Then:

Theorem 3. *There is no method by which the privileged operator can read a program C constructed using FxA instructions, nor deliberately alter it using those instructions to give an intended encrypted output.*

Proof. *The idea to consider with respect to multiplication by a constant k is:*

$$(x + h)k = (xk + h) + h(k - 1)$$

So an arbitrary '+h' may be applied both to the data x going in and the data xk coming out of a 'multiply by k' instruction if an instruction to add $h(k-1)$ is appended. I.e., replacing the fused instruction $[y \leftarrow x*k_1+k_2]_\mathcal{E}$ by $[y \leftarrow x*k_1+k'_2]_\mathcal{E}$ where $k'_2=h(k_1-1)+k_2$, supports a '+h' change in the circulating data but the change is inscrutable to the operator, who is not privy to the value of the encrypted constants $[k_2]_\mathcal{E}$ vs. $[k'_2]_\mathcal{E}$. Then the operator cannot differentiate between '+0' and '+h', so cannot know what is circulating.

A hypothetical 'divide by 2' instruction is covered like this too. It should be both preceded by and followed by 'add a constant' instructions. The 'preceded by' part can be merged with a prior FxA instruction. E.g., for h=7, the idea is

$$\lfloor(2n+8)/2\rfloor+k+3 = n+k+7 = \lfloor(2n)/2\rfloor+k+7$$
$$\lfloor((2n-1)+8)/2\rfloor+k+3 = n+k+6 = \lfloor(2n-1)/2\rfloor+k+7$$

so for both $x = 2n$ and $x = 2n - 1$

$$\lfloor(x+7+1)/2\rfloor+k+3 = \lfloor x/2\rfloor+k+7$$

showing that replacing the fused instruction $[y \leftarrow x/2+k]_\mathcal{E}$ by $[y \leftarrow (x+1)/2+k']_\mathcal{E}$, where $k'=k+3$, supports the underlying '+7' change. The code change is inscrutable to the operator, who cannot then differentiate between '+0' and '+7'.

Variants of that calculation cover arbitrary constant arithmetic right shifts, and division by a constant divisor. Two-parameter addition is covered by

$$x+h + y+h + k = x+y + k+h + h$$

showing that replacing fused instruction $[y{\leftarrow}x{+}y{+}k]_{\mathcal{E}}$ *by* $[y{\leftarrow}x{+}y{+}k']_{\mathcal{E}}$*, where* $k'{=}k{+}h$*, supports a '*$+h$*' change in the circulating data. The code change is be inscrutable to the operator, who must allow that it may be so.*

Two-parameter subtraction is similar. Two-parameter multiplication and division may draw off a '$+h$*' displacement into preceding instructions. That reestablishes in this larger context the argument of Theorem 1 and Theorem 2.* □

So running a complex FxA instruction set in a HEC processor is just as secure as HEROIC's one-instruction set is proved to be in Theorems 1 and 2.

The proof does not bear on bitwise operations, which should be composed as subroutines using the other arithmetic operations. But the exclusive or identity

$$(x\,\hat{}\,h)\,\hat{}\,(y\,\hat{}\,h)\,\hat{}\,k = x\,\hat{}\,y\,\hat{}\,(k\,\hat{}\,h)\,\hat{}\,h$$

shows that replacing an instruction $[y{\leftarrow}x\hat{}y\hat{}k]_{\mathcal{E}}$ by $[y{\leftarrow}x\hat{}y\hat{}k']_{\mathcal{E}}$, where $k'{=}k\hat{}h$, supports arbitrary interpretations '$\hat{}h$' on the circulating data.

There is support in part here for the argument of van Dijk et al. commented on in Section 1: security in the Cloud is reducible to the successful obfuscation of code. But instead of concluding it is impossible, it seems that code may be obfuscated. That can be done by encrypting the code, hiding the execution and encrypting the I/O, but for FxA code it can be done by encrypting only the data in the code and leaving the execution (on encrypted data) visible. It is not necessary to make code inaccessible, as in Ascend, nor to have instructions all look the same, as in HEROIC.

Practical problems are that every instruction carrying an encrypted datum is longer. It also ought to be decrypted first-time and then cached unencrypted within the processor for efficiency and security. A 128-bit encryption block-size implies that 32-bit instructions expand to about 160 bits each, via four data-carrying 32-bit prefixes per instruction, and the five-fold increase means that instructions must be read from memory/cache five times as quickly in order to maintain performance. Since memory accesses are the slowest thing about a processor (approx. 15ns latency in a processor with an approx. 0.5ns cycle), that is a bottleneck in practice.

The theorem establishes a cryptographic obfuscation result: knowing the code does not enable even a single bit of data to be guessed with any statistical certainty. All results are equally likely, as far as the operator may know. But that does not take into account the truism that human beings only write certain programs, so one may bet, for example, on finding an encrypted 1 as one of the constants in the program. Practical obfuscation requires a compiler that uses the alternatives that appear in the proof of Theorem 3, else all is bluff.

7 Obfuscating Compilation to FxA

To make compilation to FxA really use the instruction architecture for obfuscation, not just set the extra constant addend to (encrypted) zero every time, the compiler may use a database $D : \mathrm{DB} = \mathrm{Loc} \to \mathbb{Z}_{32}$ containing (32-bit) integer *offsets* indexed per register or memory location. As the compiler works through the source code, the offset represents by how much the data underneath the encryption is to vary from nominal at runtime at that point in the program.

The compiler also maintains a database $L : \mathrm{Var} \to \mathrm{Loc}$ of the locations (registers, memory) for the source variable placements:

$$C^L : \mathrm{DB} \times \mathrm{source\_code} \to \mathrm{DB} \times \mathrm{machine\_code}$$

The following three cases are representative of the compiler action:

Sequence: The compiler works left-to-right:

$$\begin{aligned} C^L[D_0 :\ s_1; s_2] &= D_2 :\ m_1; m_2 \\ \text{where } D_1 : m_1 &= C^L[D_0 : s_1] \\ D_2 : m_2 &= C^L[D_1 : s_2] \end{aligned}$$

Arithmetic: The opportunity for obfuscation arises at the compilation of non-control instructions. To compile a source code addition, for example, a new random offset $\mathrm{d}x' {=} D_1 Lx$ for the data in the target location Lx is invented:

$$\begin{aligned} C^L[D_0 : x{=}y{+}z] &= D_1 : [Lx{\leftarrow}Ly{+}Lz{+}i]_{\mathcal{E}} \\ \text{where } i &= D_1 Lx{-}D_0 Ly{-}D_0 Lz \end{aligned}$$

and D_1 varies from D_0 only at Lx. At runtime, data flows in to the instruction with offsets of $\mathrm{d}x{=}D_0 Lx$, $\mathrm{d}y{=}D_0 Ly$, $\mathrm{d}z{=}D_0 Lz$ respectively in Lx, Ly, Lz, and flows out with offset $\mathrm{d}x'{=}D_1 Lx$ in Lx. The locations Lx, Ly, Lz appear in the assembler code instead of the source code variables x, y, z. The $[Lx{\leftarrow}Ly{+}Lz{+}i]_{\mathcal{E}}$ representation is assembled to a machine code instruction consisting of fields

$$[\mathrm{add}], Lx, Ly, Lz, [i]_{\mathcal{E}}$$

where the '[add]' is a readable opcode, and $[i]_{\mathcal{E}}$ is encrypted.

Return: The compiler at a 'return x' statement must change the prior offset $D_0 Lx{=}\mathrm{d}x$ of Lx to a planned but randomly chosen offset $\mathrm{d}f_{\mathrm{ret}}$ at return from function f (function instances are subtyped by offsets in their formal parameters and return value). So the compiler emits an extra add instruction $[\mathbf{v0}{\leftarrow}Lx{+}i]_{\mathcal{E}}$ with target the conventional 32-bit return value register **v0** just prior to the sequence for return via the return address register **ra** to adjust the final offset:

$$\begin{aligned} C^L[D_0 : \text{return } x] = D_1 :\ & [\mathbf{v0}{\leftarrow}Lx{+}i]_{\mathcal{E}} \\ & \ldots \quad \#\text{ restore stack} \\ & [\mathrm{jr}], \mathbf{ra}\ \#\text{ jump return} \\ \text{where } i = \mathrm{d}f_{\mathrm{ret}}{-}D_0 Lx & \end{aligned}$$

The $[\mathbf{v0}{\leftarrow}Lx{+}i]_{\mathcal{E}}$ representation is assembled to machine instruction with fields

$$[\mathrm{add}], [\mathbf{v0}], Lx, [i]_{\mathcal{E}}$$

The $[i]_{\mathcal{E}}$ is an encrypted field, '[add]' is a readable opcode.

The remaining source code control constructs are treated like return. For an if statement, for example, final branch offsets are adjusted to match at the join.

Theorem 4. *The probability across different compilations that any particular runtime 32-bit value x for $[x]_\mathcal{E}$ is in location l at any given point in the program is uniformly $1/2^{32}$.*

Proof. *Suppose that before the atomic arithmetic instruction I is encountered during program compilation it is known that for locations l the value $x{+}\mathrm{d}x$ with $[x{+}\mathrm{d}x]_\mathcal{E}$ in l varies randomly across recompilations with respect to a 'standard' value x with probability $p(x{+}\mathrm{d}x{=}X){=}1/2^{32}$, and I writes value $[y]_\mathcal{E}$ in one l in particular. That y is composed of the intended functionality $f(x{+}\mathrm{d}x)$ plus an offset $\mathrm{d}y$ generated by the compilation. Then $p(y{=}Y){=}p(f(x{+}\mathrm{d}x){+}\mathrm{d}y{=}Y)$ and the latter probability is $p(y{=}Y){=}\sum\limits_{Y'} p(f(x{+}\mathrm{d}x){=}Y' \wedge \mathrm{d}y{=}Y{-}Y')$. The probabilities are independent (because I is only treated once by the compiler and $\mathrm{d}y$ is newly introduced for it), so that sum is $p(y{=}Y){=}\sum\limits_{Y'} p(f(x{+}\mathrm{d}x){=}Y')p(\mathrm{d}y{=}Y{-}Y')$. That is $p(y{=}Y){=}\frac{1}{2^{32}}\sum\limits_{Y'} p(f(x{+}\mathrm{d}x){=}Y')$. Since the sum is over all possible Y', the total of the summed probabilities is 1, and $p(y{=}Y){=}1/2^{32}$. The distribution of $x{+}\mathrm{d}x$ in other locations is unchanged. That is the only difficult case for structural induction. Loops and conditionals introduce offsets which work like $\mathrm{d}y$ above.* $\square$

Another intuition is that dy has maximal entropy, so adding it at the end of an instruction swamps any information the instruction might have introduced.

The theorem states that code is obfuscated. Having code in hand does not tell the operator which of the many compilations it might be. The data under the encryption at any point in the program can vary arbitrarily and uniformly.

Can less use be made of FxA constant addends, which are costly at runtime? No, a single arithmetic instruction without the addend would introduce a weakness: the operator has access – via a debugger, for example – to running code, so every machine instruction can be observed and experimented with. The action may be repeatedly observed to build up an encrypted arithmetic table. Then the operator might come across, for example, two encrypted additions '$a{+}b{=}c$' and '$c{+}b{=}a$', from which it may be deduced that $2b{=}0 \pmod{2^{32}}$, so $b{=}0$ or $b{=}2^{31}$.

Observing instead an FxA addition instruction only permits $2(b{+}k){=}0$ to be deduced, where k is the unknown extra addend, which says nothing about b.

8 Codicil: Encrypted RAM

Two of the HEC platforms considered in this paper cause data to be stored to memory in encrypted form (and expect to read encrypted data from memory) by virtue of the fact that data circulates in encrypted form through the processor, which is very different from a conventional CPU that has been equipped to work with encrypted memory. The latter has been suggested as early as [26], which claims it is clear 'to those skilled in the art' that an encryption/decryption unit ('codec') may be placed on the memory path to cause memory contents to be encrypted (the aim then was to defeat attacks on home media playback systems).

Vulnerabilities via the operating system aside, unencrypted memory is generally vulnerable to so-called 'cold boot' attacks [44, 22, 24] – essentially, premoved hysically freezing the memory sticks in order to retain a snapshot of transitory content – so there has historically been an incentive towards encrypted RAM.

SGX: Intel's current SGX$^{\text{TM}}$ ('Software Guard eXtensions') processor technology [2] is the modern inheritor to the tradition. It is often cited in relation to secure computation in the Cloud, because of the separation between users that it enforces. However, the underlying security mechanism is key management to restrict users to different memory 'enclaves', which may also contain encrypted data via codecs on the memory path. In consequence, SGX machines are often used [43] by cloud service providers where the assurance of safety is a selling point. Nevertheless, the assurance is founded in the customer's trust in electronics designers 'getting it right' rather than mathematical analysis and proof.

Oblivious RAM: At the component level, 'oblivious RAM' [39, 38, 36] and its evolutions [37, 34]) is often cited as a defence against dynamic memory snooping. An oblivious RAM remaps locations dynamically, remembering the aliases, so access patterns are statistically impossible to spot. It also masks the programmed accesses in a sea of independently generated random accesses. But it is no defence against an operator with a debugger, who does not care where the data is stored.

Memory in HEC platforms: Memory addressing in the clear potentially leaks information via patterns of memory accesses. Encrypted addressing defends against that and is a natural outcome of the encrypted form in which all data (addresses too) exists in a HEC platform, but it is problematic in that nondeterministic many-valued encryption leads to programs experiencing one address accessing many different locations, and in some the data has been written and in some it has not (yet) been written. The effect is called *hardware aliasing* [4].

HEROIC, in order to avoid that, uses deterministic (i.e., one-valued) encryption. That could leave the encryption vulnerable – an observer might recognise a coincidence $x+x=x$, for example, and deduce x is 0. Statistically, only about 2^{16} 16-bit arithmetic operations could be run before seeing algebraic relations like that. But in HEROIC it would be $x+k=x$, and Section 5's condition on avoiding cipherspace collisions between program constants k and runtime data x is vital for Theorem 1's guarantee that such 'give-away' observations may never occur.

But a generic KPU platform uses nondeterministic encryption, and does exhibit hardware aliasing. To work, software has to be compiled so that addresses are copied for later reuse, rather than recalculated on need [8] (it is possible to type-check existing machine code to see if it satisfies that requirement [6], and correct it). So a hierarchy of addressing characteristics may be noted here:

1)	address obfuscation	(Ascend; oblivious RAM);
2)	deterministic address encryption	(HEROIC);
3)	nondeterministic address encryption	(KPU).

Individually scattered encrypted addresses in both HEROIC and the KPU are remapped to a physically contiguous RAM address space, which requires unit granularity of the processor *translation lookaside buffer* (TLB) hardware. That does linear remapping on a first-come, first-served basis. Unit granularity is 10^3 times finer than conventional, which carries considerable performance penalty, but the remapping recovers data locality, rehabilitating cache performance.

Some 'oblivious RAM' behaviour is generated by this arrangement. Multi-valued address encryption in a KPU means that each plaintext address varies its physical location sporadically. Indeed, in both HEROIC and the KPU, the TLB may be designed to remap each address randomly before every write. In conclusion, HEC platforms are different from 'encrypted RAM' but share features.

Oblivious RAM, 'moat' electronics [27] (to mask information leaks via power consumption) and other physical protections for conventional processors [28] may also be applied on top of any HEC processor design, Ascend being an instance.

Conclusion

This paper has considered the security of homomorphic encrypted computation (HEC) against the operator as an adversary in several working HEC processor platforms. Computational completeness in conjunction with HEC creates vulnerabilities. An efficient instruction set architecture has been defined that allows code and program traces and (encrypted) data circulating in the processor to be accessible to and modifiable by the operator, while privacy for the user is formally guaranteed. Then encrypted data cannot be read by the operator, nor programs altered to give an intended result. An obfuscating compiler provably provides uniformly distributed hidden random variation for practical support.

References

1. Adida, B.: Helios: Web-based open-audit voting. In: Proc. USENIX Sec. Symp. pp. 335–348. USENIX (2008)
2. Anati, I., Gueron, S., Johnson, S.P., Scarlata, V.R.: Innovative technology for CPU based attestation and sealing. In: Proc. 2nd Int. Workshop Hardware and Arch. Support for Sec. and Privacy (HASP'13). ACM, New York (Jun 2013)
3. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) Proc. 21st Annu. Int. Cryptol. Conf. (CRYPTO'01). pp. 1–18. Adv. Cryptol., Springer (2001)
4. Barr, M.: Programming Embedded Systems in C and C++. O'Reilly & Associates, Inc., Sebastopol, CA, 1st edn. (1998), chap. 6, pp. 64–92
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proc. 3rd Conf. Innov. Theor. Comp. Sci. (ITCS'12). pp. 309–325. ACM, New York (2012)
6. Breuer, P.T., Bowen, J.P.: Typed assembler for a RISC crypto-processor. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) Proc. Int. Symp. Eng. Sec. Softw. Syst. (ESSoS'12). pp. 22–29. No. 7159 in Lect. Notes Comp. Sci., Springer (2012)
7. Breuer, P.T., Bowen, J.P.: A fully homomorphic crypto-processor design: Correctness of a secret computer. In: Proc. Int. Symp. Eng. Secure Softw. Syst. (ESSoS'13). pp. 123–138. No. 7781 in Lect. Notes Comp. Sci., Springer, Heidelberg (2013)
8. Breuer, P.T., Bowen, J.P.: Avoiding hardware aliasing: Verifying RISC machine and assembly code for encrypted computing. In: Proc. 2nd IEEE Workshop Reliab. & Sec. Data Anal. (RSDA'14). pp. 365–370. IEEE Comp. Soc., Los Alamitos (2014)
9. Breuer, P.T., Bowen, J.P.: A fully encrypted microprocessor: The secret computer is nearly here. Procedia Comp. Sci. 83, 1282–1287 (Apr 2016)

10. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: A Practical Encrypted Microprocessor. In: Callegari, C., van Sinderen, M., Sarigiannidis, P., Samarati, P., Cabello, E., Lorenz, P., Obaidat, M.S. (eds.) Proc. 13th Int. Conf. Sec. and Cryptog. (SECRYPT'16). vol. 4, pp. 239–250. SCITEPRESS, Portugal (Jul 2016)
11. Buer, M.: CMOS-based stateless hardware security module (Apr 6 2006), US Pat. App. 11/159,669
12. Conway, J.H.: Fractran: A simple universal programming language for arithmetic. In: Open Probs. Commun. & Comp., pp. 4–26. Springer, Heidelberg/Berlin (1987)
13. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Springer, Heidelberg/Berlin (2002)
14. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Proc. 29th Annu. Int. Conf. Theor. & Applicat. Cryptog. Tech. (EUROCRYPT'10). pp. 24–43. Springer, Heidelberg/Berlin (2010)
15. van Dijk, M., Juels, A.: On the impossibility of cryptography alone for privacy-preserving cloud computing. HotSec 10, 1–8 (2010)
16. El Gamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakley, G.R., Chaum, D. (eds.) Proc. 4th Annu. Conf. Adv. Cryptol. (CRYPTO'84). pp. 10–18. Springer, Heidelberg/Berlin (1985)
17. Fletcher, C.W., van Dijk, M., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: Proc. 7th ACM Scalable Trusted Comput. Workshop (STC'12). pp. 3–8. ACM, New York (2012)
18. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proc. 41st Annu. ACM Symp. Theor. Comput. (STOC'09). pp. 169–178. New York (2009)
19. Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: Proc. 30th Annu. Int. Conf. Theor. & Applicat. Cryptog. Tech. (EUROCRYPT'11), pp. 129–148. No. 6632 in Lect. Notes Comp. Sci., Springer (2011)
20. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) Proc. 33rd Annu. Cryptol. Conf. (CRYPTO'13), pp. 75–92. No. 8042 in Lect. Notes Comp. Sci., Springer, Heidelberg/Berlin (Aug 2013)
21. Graepel, T., Lauter, K., Naehrig, M.: ML confidential: Machine learning on encrypted data. In: Kwon, T., Lee, M.K., Kwon, D. (eds.) Proc. 15th Int. Conf. Inform. Sec. & Cryptol. (ICISC'12), pp. 1–21. Springer, Heidelberg/Berlin (2013)
22. Gruhn, M., Müller, T.: On the practicability of cold boot attacks. In: Proc. 8th Int. Conf. Avail., Reliab. & Sec. (ARES'13). pp. 390–397. IEEE (Sep 2013)
23. Hada, S.: Zero-knowledge and code obfuscation. In: Okamoto, T. (ed.) Proc. 6th Int. Conf. Theor. & Applicat. Cryptol. and Inform. Sec. (ASIACRYPT'00), pp. 443–457. No. 1976 in Lect. Notes Comp. Sci., Springer, Heidelberg/Berlin (2000)
24. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM 52(5), 91–98 (2009)
25. Hartman, R.: System for seamless processing of encrypted and non-encrypted data and instructions (Jun 29 1993), US Patent 5,224,166
26. Hashimoto, M., Teramoto, K., Saito, T., Shirakawa, K., Fujimoto, K.: Tamper resistant microprocessor (2001), US Patent 2001/0018736
27. Kissell, K.: Method and apparatus for disassociating power consumed within a processing system with instructions it is executing (2006), US Pat. App. 11/257,381
28. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) Proc. 19th Annu. Int. Cryptol. Conf. (CRYPTO'99), pp. 388–397. Advances Cryptol., Springer, Heidelberg/Berlin (Aug 1999)

29. Kömmerling, O., Kuhn, M.G.: Design principles for tamper-resistant smartcard processors. In: Proc. USENIX Smartcard Tech. Workshop. pp. 9–20. USENIX Association, Berkeley, CA (May 1999)
30. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: FOCS. vol. 97, pp. 364–373 (1997)
31. Lagendijk, R.L., Erkin, Z., Barni, M.: Encrypted signal processing for privacy protection: Conveying the utility of homomorphic encryption and multiparty computation. IEEE Signal Processing Magazine 30(1), 82–105 (2013)
32. Lauter, K., Naehrig, M., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proc. 3rd ACM Cloud Comput. Sec. Workshop (CCSW'11). pp. 113–124. ACM, New York (2011)
33. Lenstra, H.W., Pomerance, C.: A rigorous time bound for factoring integers. J. AMS 5(3), 483–516 (1992)
34. Liu, C., Harris, A., Maas, M., Hicks, M., Tiwari, M., Shi, E.: Ghostrider: A hardware-software system for memory trace oblivious computation. In: Proc. Int. Conf. Arch. Support for Prog. Lang. & Oper. Syst. (ASPLOS'15) (2015)
35. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation in the cloud via multikey fully homomorphic encryption. In: Proc. 44th ACM Symp. Theor. Comput. (STOC'12). pp. 1219–1234. ACM, New York (2012)
36. Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: Proc. Theor. Cryptog., pp. 377–396. Springer, Heidelberg/Berlin (2013)
37. Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., Song, D.: Phantom: Practical oblivious computation in a secure processor. In: Proc. ACM Conf. Comp. & Commun. Sec. (SIGSAC'13). pp. 311–324. ACM (2013)
38. Ostrovsky, R., Goldreich, O.: Comprehensive software protection system (Jun 16 1992), US Patent 5,123,045
39. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: Proc. 22nd Annu. ACM Symp. Theor. Comput. pp. 514–523. ACM, ACM (1990)
40. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Proc. Int. Conf. Theor. & Applicat. Cryptog. Tech. (EUROCRYPT'99). pp. 223–238. No. 1592 in Lect. Notes Comp. Sci., Springer (1999)
41. Patterson, D., Hennessy, J.: Computer Organization and Design: the Hardware/Software Interface. Morgan Kaufmann, San Mateo, CA (1994)
42. Peikert, C., Waters, B.: Lossy trapdoor functions and their applications. SIAM J. Computing 40(6), 1803–1844 (2011)
43. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: VC3: Trustworthy data analytics in the cloud using SGX. In: IEEE Symp. Sec. & Privacy. pp. 38–54 (May 2015)
44. Simmons, P.: Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In: Proc. 27th Annu. Comp. Sec. Applicat. Conf. (ACSAC'11). pp. 73–82. ACM, New York (2011)
45. Smart, N., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public Key Cryptography, pp. 420–433. No. 6056 in Lect. Notes Comp. Sci., Springer, Heidelberg/Berlin (2010)
46. Tsoutsos, N.G., Maniatakos, M.: The HEROIC framework: Encrypted computation without shared keys. IEEE Trans. CAD Integ. Circ. & Syst. 34(6), 875–888 (2015)
47. Tsoutsos, N., Maniatakos, M.: Investigating the application of one instruction set computing for encrypted data computation. In: Proc. Int. Conf. Sec., Privacy & Appl. Cryptog. Eng., pp. 21–37. Springer, Heidelberg/Berlin (2013)