# On Security in Encrypted Computing

Peter T. Breuer[1], Jonathan P. Bowen[2,4], Esther Palomar[3], and Zhiming Liu[4(✉)]

[1] Hecusys LLC, Atlanta, GA, USA
ptb@hecusys.com
[2] London South Bank University, London, UK
jonathan.bowen@lsbu.ac.uk
[3] Birmingham City University, Birmingham, UK
esther.palomar@bcu.ac.uk
[4] RISE, Southwest University, Chongqing, China
zhimingliu88@swu.edu.cn

**Abstract.** Encrypted computing is an emerging approach to security and privacy of user data on a computing system with respect to the operating system and other powerful insiders as adversaries. It is based on a processor that 'works encrypted', taking encrypted inputs to encrypted outputs while data remains in encrypted form throughout processing. An appropriate machine code instruction set is required, plus an 'obfuscating' compiler, and then the three part system provably provides cryptographic semantic security for user data, given that the encryption is independently secure. In other words, encrypted computing does not compromise the encryption. This paper presents the developing theory.

**Keywords:** Encrypted computing · Computer security · Data security

## 1 Introduction

This paper describes an emerging approach to provable security for user data against the operator, operating system and other powerful insiders in a computing system: *encrypted computing*. By that is meant that the processor takes inputs and produces outputs in encrypted form and observations via the programming interface of its internal states show only encrypted data. Our aim in this document is to project the developing theory. Engineered boundaries have fallen short as security barriers in the past, as recent attacks [14] on Intel's flagship SGX™ [1] architecture for secure computing attest. The mathematics of encrypted computing shows that, to any adversary who does not know the encryption, the feasible interpretations of a program code and its execution trace are arbitrarily many and any method of attack, whether known or unknown, deterministic or stochastic, must fail to uncover what a given bit of data is with better than the probability from guesswork (see Sect. 8). That is the definition of *cryptographic semantic security* [13], and access rights are not a consideration.


D. Naccache et al. (Eds.): ICICS 2018, LNCS 11149, pp. 192–211, 2018.
https://doi.org/10.1007/978-3-030-01950-1_12

The adversary in this setting is technically the *operator mode* of working of a suitable processor, and attacks are programs composed of the processor's machine code instructions. The operator mode 'works unencrypted' in the conventional way in the supporting processor, while the *user mode* 'works encrypted' as described in the opening to this section. Operator (also called 'supervisor') mode is synonymous with no access restrictions, whereas user mode is restricted to certain registers and areas of memory. Operator mode is the mode in which the operating system runs and a processor starts in operator mode when it is switched on, in order to load the operating system code from disk. Conventional software relies on the processor to change from user mode to operator mode and back to supply system support (e.g., disk I/O) as required, so the operator mode of working of the processor intrinsically presents difficulties as an adversary for the user mode. This document will use 'the operator' for operator mode. A malicious operating system is 'the operator', as is a human with administrative privileges, perhaps obtained by physically interfering with the boot process.

How the user gets an encryption key into the supporting processor is not the subject of this document. Diffie-Hellman hardware [7] may do key-exchange in public view to a write-only internal store, for example, without revealing the key to any observer, the operator included. A simple argument says there is not even a penalty to getting key management wrong: if (a) user B's key is still loaded internally when user A runs, then A's programs do not run correctly because the running encryption is wrong for them and A is as badly off as a spy as the operator but with less privilege, and if (b) B's key is in the machine together with B's program when A runs, then user A cannot supply appropriate encrypted inputs nor interpret the encrypted output, and is in no better position than the operator, against whom encrypted computing should already protect.

A possible scenario for an attack by the operator is where user data consists of scenes from animation cinematography being rendered in a server farm. The computer operators at the server farm have an opportunity to pirate for profit portions of the movie before release and they may be tempted. Another possible scenario is the processing in a specialised facility of satellite photos of a foreign power's military installations to reveal changes since a previous pass. If an operator (or a hacked operating system) can modify the data to show no change where there has been some, then that is an option for espionage. A *successful attack* by the operator is one that discovers the plaintext of user data or alters it to order. That is familiar in everyday situations too – for example, malware can gain operator system access and intercept the plaintext of encrypted user mail.

Note that it is not claimed here that the operator will not be able to interfere with user data at all; they can, say by writing zeros to memory or turning the machine off. What is claimed is that the operator cannot interfere so as to write in user data an intended independently defined value such as $\log \pi$ or the encryption key, or bias the likelihood of that outcome. That theory is explained here.

A medium term practical goal is a server for remote batch ('offline') computations. In that paradigm, the user compiles the program anew for each new set of (encrypted) inputs, submits the input and object code to the remote platform,

and receives back (encrypted) outputs. Theory says there will be no relation between the plaintext values beneath the encryption in the trace of one run versus that in another, so the arrangement is awkward to attack. The encryption key may be changed from run to run. However, there may be no need to change it frequently as the user's program will also offset inputs and outputs (and intermediate values) by different random amounts known only to the user beneath the encryption for each new run. The offset by different numbers each time everywhere among the plaintext values is a generator of maximal entropy for what is effectively an extra one-time coding pad beneath the encryption.

This article is organised as follows. Section 2 gives the historical context and state of the art. Section 3 sets out the components of an encrypted computing system. Section 4 shows by example what encrypted computing looks like. Security problems that generically arise from naive encrypted computing are considered in Sect. 5. Section 6 introduces theory to overcome it, first introduced in [5]. An appropriate machine code instruction set is required, and that is described in Sect. 7. An 'obfuscating' compiler is also required and that is described in Sect. 8. Section 8 shows the combination of processor, instruction set and compiler guarantees semantic security 'relative to the security of the encryption' (the hypothesis that the encryption is independently secure). The meaning of that is 'encrypted computing does not compromise encryption'.

**Notation**

Encryption of plaintext $x$ is denoted by $\mathcal{E}[x]$ or $x'$, where $\mathcal{E}$ is a one-to-many 'nondeterministic function', a function of $x$ and extra hidden variables such as padding. Decryption of ciphertext $\zeta$ is denoted by $\mathcal{D}[\zeta]$, a function, with $\mathcal{D}[x'] = x$. The key $k$ for encryption/decryption will be implicit when only one is involved, otherwise $\mathcal{E}[x, k]$ and $\mathcal{D}[\zeta, k]$. Equality (not identity) of ciphertexts $\chi = \zeta$ is defined as $\mathcal{D}[\chi] = \mathcal{D}[\zeta]$, so $x' = y'$ iff $x = y$, with $x' \neq y'$ iff $x \neq y$.

Operations on ciphertext will borrow the same names as on plaintext but in square brackets. Thus $\mathcal{E}[x_1]\,[+]\,\mathcal{E}[x_2] = \mathcal{E}[x_1 + x_2]$, meaning that $\mathcal{E}[x_1]\,[+]\,\mathcal{E}[x_2]$ may be calculated by decrypting the ciphertexts back to plaintexts $x_1$, $x_2$, adding, then encrypting again. Whether the calculation is like that or not (the encryption may already possess that property), the abstraction is applicable.

## 2 Background

In 2009 Gentry produced a fully homomorphic encryption (FHE) [10], fulfilling a prediction of Rivest et al. [26] 30 years earlier. That is an encryption in which ciphertexts can be added to add the (1-bit) plaintexts beneath, and multiplied to multiply them. In 2010 one of the present authors realised that homomorphism is a joint function of arithmetic and encryption together, and hardware can be redesigned to provide the arithmetic that makes any given encryption homomorphic with respect to it. Moreover, conditionals (not part of Gentry's

scheme) can also be handled, giving rise to computationally complete hardware-assisted encrypted computing. The proof of that was published in 2013 [4]. It followed experiments that built a model of a pipelined superscalar processor in Java (http://sf.net/p/jmips) and replaced its arithmetic logic unit (ALU), generating encrypted working (http://sf.net/p/kpu) as predicted.

From 2014 to 2016, the open source *or1ksim* simulator (http://opencores.org/or1k/Or1ksim) for the *OpenRISC* (http://openrisc.io) processor architecture was modified first to 64-bit and then 128-bit encrypted computing and cycle-accurate simulation of a complete OpenRISC compliant reduced instruction set computer (RISC) [23] 'running encrypted'. That (a) demonstrated the principle of working in a superscalar model for engineers who may not have accepted mathematical proofs and formally-oriented computer science, and also (b) explored the limits. With respect to (b), it was unknown if conventional instruction sets and processor architectures and organisation would be compatible with the idea, or how interactions with the operating system and processor interrupts would work. It became clearer, for example, that not every kind of code could run encrypted in the context – compilers and programs that arithmetically transform the addresses of program instructions (as distinct from addresses of program data) must run unencrypted because instruction addresses remained unencrypted by design, in order to prevent known plaintext attacks (KPAs) [2] on encrypted but predictable address sequences in a trace.

The existing GNU *gcc* v4.9.1 compiler (http://github.com/openrisc/or1k-gcc) and *gas* v2.24.51 assembler (http://github.com/openrisc/or1k-src/gas) ports for OpenRISC v1.1 were adapted for an encrypted instruction set (executables are the standard ELF format). Those now twice-ported compiler and utilities are at http://sf.net/p/or1k64kpu-gcc and http://sf.net/p/or1k64kpu-binutils respectively. It turns out that only the assembler, not the compiler, needs to know the encryption key. The largest application suite[1] ported to encrypted running so far for that project is 22,000 lines of C, and it and every application ported (now about fifty) has worked well. Though the target platform is 32-bit beneath the encryption, 64-bit integer and 32- and 64-bit floating point programs work well, because of code-level translations that *gcc* performs for platforms without 64-bit and floating point hardware support.

In 2015 details of the HEROIC processor for encrypted computing with Paillier-2048 encryption (of 16-bit data) were published [30]. The basic operation is a 16-bit plaintext/2048-bit ciphertext addition in 20 $\mu$s, equivalent in speed to a 25 KHz classic Pentium. The machine has a stack-based architecture. Those are different from conventional von Neumann architectures but there have been hardware prototypes [15,28] aimed at Java. A difficulty in using Paillier is that, though it is homomorphic with respect to plaintext addition, that is not mod $2^{16}$ addition, so each addition result has to be renormalised mod $2^{16}$ beneath the encryption every time, which accounts for half the cycles taken. It is done by subtracting $2^{16}$ and looking up a 'table of signs' for encrypted numbers to see if the result is negative or positive. To facilitate that, HEROIC encryp-

[1] IEEE floating point test suite at http://jhauser.us/arithmetic/TestFloat.html.

tion is one-to-one, not one-to-many. Significantly, its ISA is a *one instruction set computing* (OISC) design that has the property that the same program code and runtime trace can be interpreted with respect to the plaintext data beneath the encryption at any point in memory and in the control graph in arbitrarily many ways by any observer and experimenter who does not have the key to the encryption. That means the kind of compilation discussed in Sect. 8 would work to provably secure it, but it is not clear if HEROIC's authors have a compiler.

In 2018 the 10× faster CryptoBlaze architecture for encrypted computing, also using Paillier but with a nondeterministic component, was published [18].

At the other end of the scale a pathfinding earlier machine for encrypted computing, Ascend [9] (2012), did all its computation in unencrypted form, but with no access for the operator or operating machine while a program is running. Only the inputs and outputs were encrypted (including memory I/O) but the processor ran in 'Fort-Knox'-like isolation, matching pre-defined statistics on observables such as cache and power drain. Ascend ran RISC MIPS [24] instructions and slowed down by 12–13.5× in encrypted mode with AES-128 (rightly, only relative figures are quoted in [9]), as compared to 10–50% slowdown for the authors own recent processor models for encrypted computing, which have been measured at 104 MIPS (equivalent to a 433 MHz classic Pentium) [6] when clocked at 1 GHz, on the standard Dhrystones benchmark [32].

Physical isolation of processes plus encrypted memory has emerged several times as an idea for secure computing (e.g., [17] for secure entertainment media platforms) and success means doing it as well as Ascend. Otherwise side-channels such as cache-hit statistics [31] and power drain [19] leak information.

In that line, Intel's SGX™ ('Software Guard eXtensions') processor technology [1] is often cited, because it enforces separations between users. The mechanism is key management to restrict users to memory 'enclaves'. While the enclaves may be encrypted (encryption/decryption units lie on the memory path), that is encrypted storage, a venerable idea [16], not encrypted computing.

SGX machines are used [29] by cloud service providers where assurance of safety is a marketing point. That is founded in customers' belief in electronics designers 'getting it right' rather than mathematical analysis and proof. Engineering may leak secrets via timing variations and power use and SGX has recently fallen victim [14]. Use of SGX secure enclaves has to be written-in by the software author so it is a voluntary security device, whereas encrypted computing is an obligate security device. However, running code entirely inside an SGX enclave is running it in Ascend-style 'splendid isolation', but without Ascend's protection against statistically-based deductions from the observables. SGX does hide explicit timing information, for example, but a code can count its own instructions to retrieve an estimate.

IBM's efforts at making practical encrypted computation using very long integer lattice-based *fully homomorphic encryptions* (FHEs) based on Gentry's 2009 cipher deserve mention. The 1-bit logic operations take of the order of 1 s [11]

on customised vector mainframes with a million-bit word, about equivalent to a 0.003 Hz Pentium, but it may be that newer FHEs based on matrix addition and multiplication [12] will be faster. The obstacle to computational completeness is that which HEROIC overcomes with its 'table of signs': encrypted comparison with plain 1/0 output is needed, as well as the encrypted addition (and multiplication), but HEROIC's solution is not feasible for a million-bit encryption.

In principle, applications that require a fixed small number of multiplications can be carried out without overflowing using an FHE without the renormalisations that are their hallmark. Such schemes are called *somewhat homomorphic* encryption (SHE). Hardware assistance for a SHE based on the YASHE scheme [3] with ciphertext blocks $2^{15} \times 1228$ bits long is reported in [27]. Their 2048-core parallel hardware does ciphertext addition in 83 ns and multiplication in 59.413 ms, but that is for one bit in plaintext. That would be 32-bit plaintext addition at the speed of a 0.166 Hz classic Pentium (counting 3 exclusive or gates and 3 and gates per 1-bit full adder, taking 6 s for one 32 bit addition).

The slow speeds of even hardware-assisted SHE schemes emphasise how relatively fast the recent general purpose processors for encrypted computing are. The price is a secret embedded in the hardware, as with a Smartcard.

## 3 Encrypted Computing Systems: Overview

This section briefly recapitulates encrypted computing systems. They consist of:

(i) **A processor that 'works encrypted'** in user mode, with encrypted inputs, encrypted outputs, and encrypted intermediate states, but which 'works unencrypted' in operator mode.

If user data were not in encrypted form throughout, then the operator, having full access, could read it and write it to order, so this kind of processor is needed.

(ii) **A machine code instruction set** that prevents 'algebraic' attacks via certain 'chosen instructions' that conventional instruction sets contain.

For example, the conventional instruction that performs $x' \,[\,/\,]\, x'$, produces an encrypted 1 from any encrypted user datum $x'$ the operator cares to copy.

(iii) **An 'obfuscating' compiler** that smooths out statistical biases that may be present in machine code.

Else a program would contain human biases such as low numbers like 0,1,... in loop counts, which could be used in a statistically-based dictionary attack.

The following axioms from [5] refine the hardware requirements (i), (ii):

**Axioms**

(1) Each instruction's action is a black box. (i)
(2) Each instruction is observed to read and write data in encrypted form. (i)
(3) Arithmetic instructions embed encrypted constants, adjustments to which may be made to accommodate any planned offsets in inputs to and output from the instruction (see Sect. 7). (ii)
(4) There are no collisions possible between the encrypted constants embedded in some instructions and the ciphertext that the processor writes to and reads from registers and memory. (ii)

How (1) and (2) are achieved is a question of the hardware. It is failure of (1) that the Intel (and likely other manufacturers') vulnerability exploits in the recent Meltdown [21] and Spectre [20] attacks. There, speculative execution brings data into cache that remains visible even though the instructions are aborted, so 'nothing' leaves a trace. One of the conceptually simplest ways of achieving (2) is to use an encryption that permits arithmetic to be done without decrypting and re-encrypting: a *homomorphic* encryption. Some processors for encrypted computing have used homomorphic encryptions, as described in Sect. 2.

Certain classical processor features contradict (2) when 'observe' is understood to mean testing the processor state by any programmatic means, not only reading a register, and are to be avoided in designs. A machine code 'set if equal' instruction that compares two ciphertext inputs and sets a status flag if the plaintexts are equal would be mistaken instruction set design. The instruction output (the status flag) in that case would not be in encrypted form, as required by (2). It is also not in any register, but it could be tested with a following 'branch if set' instruction, because the branch is seen to be taken or not taken as the (inaccessible) status flag is set or not set. That makes a classical 'set/test-flag' style of processor instruction set design inappropriate. Yet the OpenRISC standard specifies that style of instruction set and so our own prototype processors step back to an earlier MIPS style of RISC design for branch instructions. Our own design's branch instructions do not test a status flag but compare (less than, equal to, etc.) register contents and branch or do not branch on the result as determined in conjunction with extra encrypted instruction bits (see Sect. 7). That cannot be used for binary search to determine a value by virtue of (3,4).

The axiom (3) is a feature of an appropriate instruction architecture (Sect. 7), while (4) may be achieved via disjoint paddings beneath the encryption.

A fifth axiom is sometimes needed. It extends (2) to allow testing by means of externally known facts, not only the processor's programming interface:

(5) There are no sources of ciphertext whose plaintext is known independently.

That avoids known plaintext attacks. It would contradict (5) to design-in a read-only register that holds the known processor stepping number (encrypted).

The axiom carries over to statistics too: all registers should feasibly contain anything at start-up, with equal probability. A classical RISC read-only **zer** register that contains zero (encrypted) for user mode would contradict that, so cannot be.

## 4 What Does Encrypted Computing Look Like?

Encrypted running is illustrated in Table 1, where the same program has been compiled twice, and the resulting machine codes have been run (the two instances are top, vs. bottom in the table). Each time, the compiler has embedded different (encrypted) constants in the machine code (disassembly at left). As a result different encrypted values appear throughout the execution traces (right), but the decrypted result (boxed) is nevertheless the same.

**Table 1.** Program codes and execution traces of exactly the same form may have encrypted data whose plaintext is arbitrarily different at any point yet get the same result.

1st code fragment

| *addr.* | *instruction disassembly* |
|---|---|
| 96C | sub sp sp zer $\mathcal{E}$[ -471185111] |
| 980 | jal A |
| | |
| 984 | add t0 v0 zer $\mathcal{E}$[ -236230946] |
| 998 | sub sp sp zer $\mathcal{E}$[-1219116768] |
| 9AC | lw a0 $\mathcal{E}$[1219116768](sp) |

1st fragment's trace

| *addr.* | *update* |
|---|---|
| 96C | sp ← $\mathcal{E}$[-3412890104] |
| 980 | ra ← 984 |
| ... | |
| 984 | t0 ← $\mathcal{E}$[ -236230942] |
| 998 | sp ← $\mathcal{E}$[-1219116896] |
| 9AC | a0 ← $\mathcal{E}$[ 1 ] |

2nd code fragment

| *addr.* | *instruction disassembly* |
|---|---|
| 96C | sub sp sp zer $\mathcal{E}$[ 1528657211] |
| 980 | jal A |
| | |
| 984 | add t0 v0 zer $\mathcal{E}$[-1112987554] |
| 998 | sub sp sp zer $\mathcal{E}$[ -275939886] |
| 9AC | lw a0 $\mathcal{E}$[275939886](sp) |

2nd fragment's trace

| *addr.* | *update* |
|---|---|
| 96C | sp ← $\mathcal{E}$[ -178928721] |
| 980 | ra ← 984 |
| ... | |
| 984 | t0 ← $\mathcal{E}$[-1112987550] |
| A98 | sp ← $\mathcal{E}$[ -275940014] |
| 9AC | a0 ← $\mathcal{E}$[ 1 ] |

LEGEND

Encrypted: $\mathcal{E}$[x], x′ (Same) program point and storage place
Registers: pc,ra,sp,zer,t0,v0,a0 Content: *pc*, *ra*, *sp*, *zer*, *t0*, *v0*, *a0*

INSTRUCTION SEMANTICS

sub x y z $k'$ : $x \leftarrow y\,[-]\,z\,[+]\,k'$ jal $a$ : $ra \leftarrow pc + 4;\ pc \leftarrow a$ $\mathcal{E}[x]\,[\circ]\,\mathcal{E}[y] = \mathcal{E}[x \circ y]$
add x y z $k'$ : $x \leftarrow y\,[+]\,z\,[+]\,k'$ lw x $k'(y)$ : $x \leftarrow \text{memory}[\![y\,[+]\,k']\!]$

The 'trick' is that the compiler creates code that at runtime produces encrypted values whose plaintext values are *offset* from the nominal value all the

way through the calculation. The offsets are different (and randomly generated) for each point in the program control graph per each location in memory. For illustration here, the final offset in register **a0** has been set at 0, but ordinarily the final offset is also randomly generated, albeit known to the user.

## 5 Vulnerabilities of Naive Encrypted Computation

Being able to run arbitrary computable functions is dangerous in principle because an adversary might use the encrypted computations to subvert the encryption. For example, 32-bit 2 s complement arithmetic is used in all modern computing. In that, repeated doubling of anything gives encrypted zero. I.e.:

$$\mathcal{E}[x][+]\ldots[+]\mathcal{E}[x] = \mathcal{E}[x+\ldots+x] = \mathcal{E}[2^{32}x \mod 2^{32}] = \mathcal{E}[0].$$

That opens the encryption to a known plaintext attack. The adversary can obtain encryptions of 0 by forcing the processor to add any initial datum $\mathcal{E}[x]$ in register $r$ to itself 32 times, using its own machine code addition instruction:

$$\underbrace{\textbf{add}\ r\ r\ r}_{r\leftarrow r\,[+]\,r};\ldots;\underbrace{\textbf{add}\ r\ r\ r}_{r\leftarrow r\,[+]\,r};$$

Using multiplication, choosing a random ciphertext has a 50% chance of picking an odd number plaintext and then repeated self-multiplication gives an encrypted 1, by Fermat's Little Theorem[2]:

$$\mathcal{E}[x][*]\ldots[*]\mathcal{E}[x] = \mathcal{E}[x*\ldots*x] = \mathcal{E}[x^{2^{31}} \mod 2^{32}] = \mathcal{E}[1]$$

Self-multiplying an even number gives encrypted zero, but half the time an encrypted 1 is obtained, and a 50% success rate beats $1/2^{32}$ odds from guessing.

Using division, an adversary can get an encrypted 1 from any datum that is not an encrypted zero, which is a near certainty among all the encrypted data passing through the machine, via

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

A subroutine for 64-bit division on a 32-bit platform is a place where one would find an encrypted 1 as a program constant or an extra parameter to the subroutine at each application. In any case, a dictionary attack on all the constants in the code should encounter an encrypted 1 among them. Guess which and, by

[2] Fermat's Little Theorem is $a^{\phi}=1 \mod n$, where $a$ is coprime to $n$ and $\phi$ is the size of the multiplicative group of integer residues mod $n$, being the number of residues that are coprime to $n$. It is needed here in the form $a^{\phi} = 1 \mod 2^n$, where $a$ is odd, i.e., coprime to $2^n$. Exactly half the numbers less than $2^n$ are odd, i.e., coprime to $2^n$, and they form the multiplicative group mod $n$. So $\phi$ is $2^{n-1}$ and the theorem says $a^{2^{n-1}}=1 \mod 2^n$. The better-known special form is $a^p = a \mod p$, $p$ prime.

**Table 2.** Code and trace may be interpreted in different ways with respect to the plaintext data by an observer who cannot read the encryption.

| 1st code fragment | 1st trace | 2nd code fragment | 2nd trace |
|---|---|---|---|
| *addr. instruction* | *addr. update* (x = $\mathcal{E}$[0]) | *addr. instruction* | *addr. update* (x = $\mathcal{E}$[7]) |
| 0 A:if x[<]$\mathcal{E}$[1] goto B | 0 A: | 0 A:if x[<]$\mathcal{E}$[8] goto B | 0 A: |
| 1 x←x[-]$\mathcal{E}$[1] | ↓ | 1 x←x[-]$\mathcal{E}$[1] | ↓ |
| 2 goto A | | 2 goto A | |
| 3 B:x←x[+]$\mathcal{E}$[1] | 3 B: x←$\mathcal{E}$[1] | 3 B:x←x[+]$\mathcal{E}$[1] | 3 B: x←$\mathcal{E}$[8] |
| 4 if x[<]$\mathcal{E}$[1] goto B | 4 (x = $\mathcal{E}$[1]) | 4 if x[<]$\mathcal{E}$[8] goto B | 4 (x = $\mathcal{E}$[8]) |

LEGEND
Encrypted: $\mathcal{E}$[x] $\quad \mathcal{E}[x]\,[\circ]\,\mathcal{E}[y] = \mathcal{E}[x \circ y] \quad \mathcal{E}[x]\,[R]\,\mathcal{E}[y] = x\,R\,y$

repeated addition of encrypted 1s, an adversary may first build all powers of 2 and then build the encryption of any desired number $K$ from its binary code via

$$\mathcal{E}[2^{k_1}]\,[+]\ldots[+]\,\mathcal{E}[2^{k_j}] = \mathcal{E}[2^{k_1} + \ldots + 2^{k_j}] = \mathcal{E}[K].$$

Then, if an arithmetic order comparator instruction is available on the platform, any encrypted number could be decrypted by comparing it with an encryption of each 32-bit integer $K$ in turn[3]. Decryption goes even faster deducing the binary digits one by one, comparing and subtracting (encrypted) $2^k$ when $\mathcal{E}[2^k][\leq]\mathcal{E}[x][<]\mathcal{E}[2^{k+1}]$ is detected by a conditional branch instruction in the machine (a machine code conditional branch on $\mathcal{E}[x][\leq]\mathcal{E}[y]$ detects if $x \leq y$ then jumps to a designated instruction, just like a **goto** in a higher level language).

The vulnerabilities above apply to any naive system for *arbitrary* computation. It must have comparator instructions in order to trigger branch jumps. In contrast, finite calculation systems can produce the 1/0 result $b$ of a comparison in encrypted form, and the final ciphertext values $\mathcal{E}[x_1]$ and $\mathcal{E}[x_0]$ of variable $x$ from both branches after the comparison are combined in $\mathcal{E}[x_1 * b + x_0 * (1 - b)]$. That is not an option in a system for unbounded computation, which must report the comparison in 1/0 format so the electronics can execute only one branch.

In summary, to *write* an encrypted number to order on a naively constructed platform, 'just addition and multiplication' will do, with 50% certainty. *Reading* requires a comparator too. If the encryption itself is even partially homomorphic (i.e., some encrypted operations can be done without access to the encryption key), the processor is not even needed. So there is a case to answer as to security.

## 6 Secure Encrypted Computing

There are ways of running arbitrary encrypted computations securely. Consider for the moment that the machine code has only instructions *addition of a constant* $y \leftarrow \mathcal{E}[\mathcal{D}[x]+k]$ and branches based on *comparison with a constant* $\mathcal{D}[x] < K$,

[3] Rass in [25] has recently independently called this a 'chosen instruction' attack.

for registers $x$, $y$[4]. Those suffice for any computation, encrypted[5]. Consider program $C$ *using only those two instructions*. By a 'method of observation' understand a deterministic process, based on observing what a running user program does from step to step and making deductions from what is observed – its *trace* $T$. The trace details the sequence of instructions executed, with their addresses, and what register and memory locations each instruction reads and writes and with what values. Assume *the operator cannot already read the encryption*. Then:

**Theorem 1** *No method of observation exists by which the operator (who does not possess the key) may decrypt output from* $C$.

The argument is illustrated by the program in this language that sets x=$\mathcal{E}[1]$, rendered at left in Table 2. There is no single statement of the language that will suffice. The code first loops until x is 'not too large,' then loops until it is 'not too small.' Exit at B is with x = $\mathcal{E}[1]$ exactly, no matter the value at entry at A. The trace with x = $\mathcal{E}[0]$ on entry and x = $\mathcal{E}[1]$ on exit is shown alongside. The right half of Table 2 shows the same code in which branch comparison constants (red) have been changed by +7 beneath the encryption. That admits a trace of exactly the same form but with plaintext numbers beneath the encryption that are +7 more than before. Since it is feasible, it is what happens, as computation is deterministic. So there are two possible interpretations of the codes and traces in Table 2 to an observer who does not already know the encryption. The evidence presented to the observer's method is the same both times in the observer's own terms: the codes and the traces 'look the same', the only differences lying in encrypted constants that by hypothesis the observer cannot read. One may suppose that the codes and traces are short enough that no ciphertext is repeated twice, so those encrypted values that do appear serve as no more than different labels for the same unknowns and have no more significance than that. If the observer has a method for getting at the plaintext value beneath the encryption then it must give the same answer in both cases. Yet the observer's method must be wrong in one case, because the numbers beneath the encryption all differ by 7. So the method does not exist. The formal argument is simply that:

*Proof (Theorem 1).* Change $C$ to $D$ by changing all the constants $\mathcal{E}[K]$ in comparison instructions to $\mathcal{E}[K+7]$. That permits a trace in which all data takes values not $\mathcal{E}[x]$ but $\mathcal{E}[x+7]$ at every point. The addition instructions, which are unaltered in $D$, instead of taking $\mathcal{E}[x]$ to $\mathcal{E}[x+k]$ now take $\mathcal{E}[x+7]$ to $\mathcal{E}[x+7+k]$. The observer's hypothetical method is not sensitive to the change as the observer cannot read the encryption, so the method must give the wrong answer either in the trace of $C$ or in that of $D$ about a value beneath the encryption. □

[4] To help the reader over a 'notation gap', $y \leftarrow \mathcal{E}[\mathcal{D}[x]+k]$ is written here for $y \leftarrow x\,[+]\,k'$.

[5] A practitioner's proof of the computational completeness of the instructions $y \leftarrow x+k$ and if $x<K$ ... is the mathematician J.H. Conway's well-known *Fractran* programming language [8], in which those are the only instructions. Attention in the computer hardware community may have been first drawn to the fact by [24].

*Remark 1.* The argument shows that the same code and trace may differ independently at every point beneath the encryption to the maximum extent possible. An assignment instruction $x \leftarrow \mathcal{E}[\mathcal{D}[y] + k]$ may be changed to account for an arbitrarily chosen offset $c$ (instead of $+7$) in the incoming value $y$ beneath the encryption and generate an arbitrarily chosen offset $d$ (instead of $+7$) in the outgoing value $x$ by rewriting the instruction to $x \leftarrow \mathcal{E}[\mathcal{D}[y] + k - c + d]$. □

## 7 Instruction Architecture for Encrypted Computing

What makes Theorem 1's proof work is the following:

**Lemma 1.** *Every atomic instruction's inputs $\mathcal{E}[x_1]$ and outputs $\mathcal{E}[x_0]$ may be shifted by constants to $\mathcal{E}[x_1 + k_1]$ and $\mathcal{E}[x_0 + k_0]$ respectively, by means of constants embedded (encrypted) in the instruction, for arbitrary $k_0$, $k_1$.*

That is merely a formal expression of axiom (3) of Sect. 1.

Designing a complete instruction set to comply with (3) requires careful choices to allow the processor to function with full coverage and to work quickly while compilation remains uncomplicated, also consideration of physical restrictions (instruction length, field sizes, opcode map, etc.). HEROIC's minimalistic instruction set complies, but is not suited to efficient compilation. We call any compliant instruction set a *fused anything and add* (FxA) instruction set[6] because the natural form of a compliant arithmetic instruction semantics is

$$\begin{aligned} x \leftarrow\ & \mathcal{E}[(\mathcal{D}[y] - k_1)\Theta(\mathcal{D}[z] - k_2) + k_0] \\ =\ & (y\,[-]\,k_1')\,[\Theta](z\,[-]\,k_2')\,[+]\,k_0' \end{aligned}$$

with binary operator $\Theta$ (for example, $\Theta$ may be multiplication), registers $x, y, z$.

Our own processor's instruction set for encrypted working is shown in Table 3. It bears a likeness to OpenRISC's instruction set and RISC in general, in that there is one memory load/store instruction and the rest of the instructions use registers, but 'RISCiness' stops at the increased instruction lengths. The comparison operations also contain an extra bit beneath the encrypted field that says if branch happens on success or on failure, as per axiom (3). Then:

**Theorem 2** *There is no method by which the privileged operator can read runtime data from a program $C$ constructed using instructions in Table 3.*

That is by using Lemma 1 in the proof of Theorem 1 for all arithmetic instructions of Table 3[7]. It also follows that interfering and experimenting with the program to substitute a different value for the returned result does not work:

[6] 'Addition of a constant' is not the only option. Bitwise XOR (exclusive OR) with a constant can be used, or 'multiplication by a prime and addition of a constant'. The most general possibility is to replace a conventional instruction $x \leftarrow f(y)$ by $x \leftarrow f(y \cdot k_1^{-1}) \cdot k_2$, where $\_ \cdot \_$ is the operation of a mathematical group and $\_^{-1}$ is the group inverse operation. For simplicity, addition is used throughout this paper.
.

[7] Proofs of results stated but not proved in the text are supplied in the Appendix.

**Table 3.** Machine code instruction set for encrypted working.

| *op.* | *fields* | *mnem.* | *semantics* |
|---|---|---|---|
| add | $r_0\, r_1\, r_2\, k'$ | add | $r_0 \leftarrow r_1\ [+]\ r_2\ [+]\ k'$ |
| sub | $r_0\, r_1\, r_2\, k'$ | subtract | $r_0 \leftarrow r_1\ [-]\ r_2\ [+]\ k'$ |
| mul | $r_0\, r_1\, r_2\, k'_0\, k'_1\, k'_2$ | multiply | $r_0 \leftarrow (r_1\ [-]\ k'_1)\ [*] (r_2\ [-]\ k'_2)\ [+]\ k'_0$ |
| div | $r_0\, r_1\, r_2\, k'_0\, k'_1\, k'_2$ | divide | $r_0 \leftarrow (r_1\ [-]\ k'_1)\ [/] (r_2\ [-]\ k'_2)\ [+]\ k'_0$ |
| ... | | | |
| mov | $r_0\, r_1$ | move | $r_0 \leftarrow r_1$ |
| ble | $j\ r_1\, r_2\, k'$ | branch | if $r_1 [\leq] r_2\ [+]\ k'$ then $pc \leftarrow pc+j$ |
| bge | $j\ r_1\, r_2\, k'$ | branch | if $r_1 [\geq] r_2\ [+]\ k'$ then $pc \leftarrow pc+j$ |
| ... | | | |
| b | $j$ | branch | $pc \leftarrow pc + j$ unconditionally |
| sw | $k'(r_1)\ r_2$ | store | $\mathrm{mem}[\![r_1\ [+]\ k']\!] \leftarrow r_2$ |
| lw | $r_1\ k'(r_2)$ | load | $r_1 \leftarrow \mathrm{mem}[\![r_2\ [+]\ k']\!]$ |
| jr | $r$ | jump | $pc \leftarrow r$ |
| jal | $j$ | jump | $ra \leftarrow pc + 4;\ pc \leftarrow j$ |
| j | $j$ | jump | $pc \leftarrow j$ |
| nop | | no-op | |

LEGEND

| | | | | | |
|---|---|---|---|---|---|
| r | – register indices | $k$ | – 32-bit integers | pc | – prog. count reg. |
| $j$ | – program count or incr. | '$\leftarrow$' | – assignment | ra | – return addr. reg. |
| $\mathcal{E}[x]$,$x'$ | – encrypted val. | $\mathcal{E}[x]\,[\circ]\,\mathcal{E}[y] = \mathcal{E}[x \circ y]$ | | $\mathcal{E}[x]\,[R]\,\mathcal{E}[y] = x\ R\ y$ | |

**Corollary 1.** *There is no method by which the operator can alter program $C$ using other or the same instructions to get an intended output (encrypted).*

The reason is that the program built by the operator to give the intended output cannot be built, by Theorem 2, because the output is readable, as it is known what it decrypts to (this lawyering stands in for a near repeat of the same proof).

**Example** (Theorem 2, Corollary 1). Take the encryption in the machine to be AES with key $k$, so encryption is $x' = \mathrm{AES}(x, k)$ for plaintext $x$ and ciphertext $x'$. Then there is a program $C$ that decrypts (encrypted) input data, though the whole program runs in the encrypted computing environment. It is the AES decryption routine, compiled encrypted. Suppose $x'$ decrypts to $x$, and $x''$ is the encryption of $x'$, $k'$ the encryption of $k$. Then $C(x'', k') = x'$ by definition, because the unencrypted program takes $x'$ and $k$ to $x$. That is $C(y', k') = y$, as claimed, on choosing $y = x'$. □

Fortunately, the theorem prohibits the adversary building a program that outputs the encrypted encryption key $k'$, because with it and program $C$ of the example the adversary would obtain the encryption key in the clear, via $C(k', k') = k$.

*Remark 2.* The program $C$ of the example that does decryption cannot be intentionally built by an adversary (this is proved in the Appendix).

**Example** (Corollary 1). The program that sets x=$\mathcal{E}$[1] at left in Table 2 cannot be intentionally built by the operator. Trying for '1' the operator may instead get the program at right in the table, which produces '8' (encrypted). □

There is need and potential (see Remark 1) for obfuscation here. Human beings only write certain programs, and an adversary may bet on an encrypted 1 being among the data, enabling the 'chosen instruction' attack of Sect. 5.

## 8 Obfuscating Compilation

For effective and useful 'obfuscation' in this context, plaintext data beneath the encryption should be varied from the nominal value at each of the up to $m + 32$ storage locations accessed by the program ($m$ memory locations and 32 registers) at each of the $N$ instructions of the program. A compiler can do that by varying the encrypted constants embedded in the instructions, by axiom (3). The idea is for the compiler variations to hide any human biases. Maximal noise applied by the compiler across different compilations swamps any other signal.

Let MC be the type of machine code, consisting of a sequential list of 'FxA-compliant' instructions, as for example in Table 3, and let Expr be the type of expressions, and let Off be the type of integer 'offsets'. The approach our own compiler takes is to invent and aim for a particular runtime offset from nominal:

$$[\![-]\!]^r :: \mathrm{Expr} \to (\mathrm{MC}, \mathrm{Off})$$

where $r$ is the processor register that the value of the expression is to appear in. That is, the result of compiling an expression $e$ is

$$[\![e]\!]^r = (mc, \Delta e)\,.$$

The value $e+\Delta e$ beneath the encryption will be produced in register $r$ at runtime by running the code $mc$, where $\Delta e$ has been freely chosen at compile time. That is, let $s(r)$ be the content of register $r$ in state $s$ of the processor at runtime. The machine code $mc$ emitted is designed to have operational semantics (Table 3):

$$s_0 \overset{mc}{\rightsquigarrow} s_1 \text{ where } s_1(r) = \mathcal{E}[e + \Delta e] \qquad (\star)$$

An offset $\Delta e = 0$ means the result will be the nominal value. Compilation for ($\star$) is described in detail in [5]. The upshot is that independently chosen, arbitrary offsets $\Delta e$ generated by the compiler are induced at *runtime* in the plaintext values written to every register and memory location, differing per point in the program control flow graph. The following lemma is proved in [5]:

**Lemma 2.** *The obfuscating compiler creates object codes from the same source code that are identical apart from embedded (encrypted) constants. The runtime traces are also identical apart from the ciphertext data values read and written, such that, for any particular plaintext 32-bit value $x$, the probability across different compilations that $\mathcal{E}[x]$ is in register or memory location $l$ at any given point in the trace is uniformly $1/2^{32}$, independently to the maximum extent permitted by copy instructions and loops in the code.*

The proviso is because a plain copy ('mov') instruction always has precisely the same input and output, and a loop means the variations introduced by the compiler must be the same at the beginning as at the end of the loop.

Source code for the compiler, assembler, linker, virtual machine, etc., may be downloaded from http://sf.net/projects/obfusc. The compiler currently covers all of ANSI C, and most GNU extensions except computed gotos.

In order to support arrays, pointers `p` must be declared together with a fixed 'memory zone' into which they point, thus:

```
int A[100];
restrict A int *p;
```

The **restrict** A means that the pointer never points outside the memory zone A. The compiler does not know where an unrestricted pointer will point at runtime and this declaration tells it to use an offset $\Delta$`A` pertaining to zone A at that point in the program for the pointer. Each write through `p` should change $\Delta$`A`, so the compiler accompanies it with writes to the rest of `A` to reset the other entries too. That is computationally inefficient, but cryptographically necessary. Oblivious RAM (ORAM) [22] does the same, but in hardware. In practice the processor will do the writes to memory asynchronously via the ubiquitous 'write-back' cache of contemporary processor technology, so the performance penalty is bandwidth, not latency (but a vector write instruction would be helpful).

Recalling Goldwasser & Micali's definition [13] that 'semantic security' is inability to guess a designated bit with any success above chance, Lemma 2 implies:

**Theorem 3** *Runtime user data beneath the encryption is* semantically secure *against the operator for FxA code compiled by the obfuscating compiler.*

The threat alone that the code has been compiled by an obfuscating compiler might be sufficient for the theorem, as that establishes the domain of possible variations that must be considered. But despite the look of it, the theorem is not a strong statement. It should be understood as saying that computation in an encrypted computing system does not reduce the security from the encryption.

## 9 Conclusion

This paper intends to bring encrypted computing to the attention of the security community as a technology that potentially safeguards user data against a classically all-powerful operator, operating system and other insiders as adversaries. This paper has dealt with theoretical aspects. With the appropriate instruction set and an 'obfuscating' compiler, it is shown here that user data cannot be determined by an adversary via any deterministic or stochastic method with any success above chance, provided the encryption used is independently secure. In other words, encrypted computing does not compromise encryption.

**Acknowledgments.** Zhiming Liu thanks the Chinese NSF for support from research grant 61672435, and Southwest University for grant SWU116007. Peter Breuer thanks Hecusys LLC for continued support in KPU research and development.

## Appendix: Proofs of results

*Proof (Corollary 1).* Suppose for contradiction that the operator builds a new program $D{=}f(C)$ that returns $\mathcal{E}[y]$. Then its constants $\mathcal{E}[k]$ are found in $C$ and its constants $\mathcal{E}[K]$ likewise, because $f$ has no way of arithmetically combining them (the disjoint subspaces condition (4) on runtime encrypted data and encrypted

program constants means they cannot be combined arithmetically in the processor and the operator does not have the encryption key). Theorem 1 says the operator cannot read output $\mathcal{E}[y]$ of $D$, yet knows what it is. Done by contradiction. □

*Proof (Theorem 2).* Program $C$ is constructed using arbitrary instructions from Table 3 compliant with (3). One may construct a modified code $D$ (see below) that looks the same as $C$ to the adversary who cannot read the encryption, as well as possessing a runtime trace $U$ that looks the same as the original trace $T$ to the adversary, differing only in the cipherspace values read and written. The argument is the same as for Theorem 1 (and Corollary 1): In the given program $C$, every binary arithmetic instruction necessarily has semantics of the form (the $r_i$ are registers)

$$r_0 \leftarrow \mathcal{E}[(\mathcal{D}[r_1]-k_1)\,\Theta(\mathcal{D}[r_2]-k_2)+k_0]$$

in order to comply with (3), and it can be adjusted for $D$ via its embedded constants $\mathcal{E}[k_i]$ to accommodate every data value passing through registers and memory to be +7 more beneath the encryption than it used to be in $C$, as argued in the proof of Theorem 1 and Corollary 1. The change is from $k_i$ to $k_i'{=}k_i + 7$. Similarly, every branch instruction in $C$ necessarily has a test of the form $(\mathcal{D}[r_1]-k_1)R(\mathcal{D}[r_2]-k_2)$ in order to comply with (3). It is changed in $D$ to $(\mathcal{D}[r_1]-k_1')R(\mathcal{D}[r_2]-k_2')$ with $k_i' = k_i + 7$. Then the branch goes the same way at runtime in trace $U$ for $D$ as it did originally in trace $T$ for $C$. Unconditional jump instructions are not altered.

The outcome is a trace $U$ that is the same as $T$ modulo the cipherspace values read and written, which by hypothesis cannot be read by the adversary. Those differ by 7 under the encryption in $U$ from the originals in $T$. Code $D$ looks the same too, modulo the embedded encrypted constants, which also cannot be read by the adversary. Therefore, as in the proof of Theorem 1, a method $f(C,T)$ for decryption must give the same result as $f(D,U)$, yet the answers are different by 7 in the two cases, so the method $f$ cannot exist. □

*Proof (Lemma 2).* Consider the arithmetic instruction $I$ in the program. Suppose that by fiddling with the embedded constants in the other instructions in the program it is already possible for all other locations $m$ other than that written by $I$ and at all other points in the program to vary the value $x_m = x + \Delta x$, where $\mathcal{E}[x_m]$ is stored in $m$, randomly and uniformly across compilations, taking advantage of the instruction set as the compiler described in the text does. Let $I$ write value $\mathcal{E}[y]$ in location $l$. By the axiom (3) $I$ has a parameter $\mathcal{E}[k]$ that may be tweaked to offset $y$ from the nominal result $f(x + \Delta x)$ on its input $x + \Delta x$ by an amount $\Delta y$. The compiler chooses $k$ with a distribution such that $\Delta y$ is uniformly distributed across the possible range. The instructions in the program that receive $y$ from $I$ may be adjusted to compensate for the $\Delta y$ change by changes in their controlling parameters. Then $p(y = Y) = p(f(x+\Delta x)+\Delta y = Y)$ and the latter probability is $p(y = Y) = \sum_{Y'} p(f(x + \Delta x) = Y' \wedge \Delta y = Y - Y')$. The probabilities are independent (because $\Delta y$ is newly introduced just now),

so that sum is $p(y = Y) = \sum_{Y'} p(f(x + \Delta x) = Y')p(\Delta y = Y - Y')$. That is

$$p(y{=}Y){=}\frac{1}{2^{32}} \sum_{Y'} p(f(x{+}\Delta x){=}Y').$$

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 data $\mathcal{E}[x_m]$ in other locations $m$ is unchanged. Done by a structural induction on the machine code program. □

*Proof (Theorem 3).* Consider a probabilistic method $f$ that guesses for a particular runtime value beneath the encryption 'the top bit $b$ is 1, not 0', with probability $p_{C,T}$ for program $C$ with trace $T$. The probability that $f$ is right is

$$\mathrm{p}((b_{C,T}{=}1 \text{ and } f(C,T){=}1) \text{ or } (b_{C,T}{=}0 \text{and} f(C,T){=}0))$$

Splitting the conjunctions, that is

$$\begin{aligned} &\mathrm{p}(b_{C,T}{=}1)\,\mathrm{p}(f(C,T){=}1 \,|\, b_{C,T}{=}1) \\ +\ &\mathrm{p}(b_{C,T}{=}0)\,\mathrm{p}(f(C,T){=}0 \,|\, b_{C,T}{=}0) \end{aligned}$$

But the method $f$ cannot distinguish the compilations it is looking at as the codes and traces are the same, modulo the (encrypted) values in them, which the adversary cannot read. The method $f$ applied to $C$ and $T$ has nothing to cause it to give different answers but incidental features of encrypted numbers and its internal spins of a coin. Those are *independent* of if the bit $b$ is 1 or 0 beneath the encryption, supposing the encryption is effective. So

$$\begin{aligned} \mathrm{p}(f(C,T) = 1 \,|\, b_{C,T} = 1) &= \mathrm{p}(f(C,T) = 1) = p_{C,T} \\ \mathrm{p}(f(C,T) = 0 \,|\, b_{C,T} = 0) &= \mathrm{p}(f(C,T) = 0) = 1 - p_{C,T} \end{aligned}$$

By Lemma 2, 1 and 0 are equally likely across all possible compilations $C$, so the probability $f$ is right reduces to

$$\tfrac{1}{2}\ p_{C,T} + \tfrac{1}{2}\ (1 - p_{C,T}) = \tfrac{1}{2}$$

since $\mathrm{p}(b_{C,T}{=}1) = \mathrm{p}(b_{C,T}{=}0) = \frac{1}{2}$. □

**Corollary 2.** *There is no method by which the operator can build a program $C$ that gives an output $\mathcal{E}[y]$ where $y$ is confined to an independently defined proper set $Y$ of possibilities, not even stochastically with a probability higher than $|Y|/2^{32}$.*

*Proof.* The proof of Theorem 2 and Corollary 1 may be repeated, confining $y$ to $Y$, or use Theorem 3, since its 'probabilistic method $f$' includes constructing a program. □

*Proof. (Remark 2).* The structure of the code of the AES decryption routine is known to the operator. By Corollary 1 the operator cannot construct the (encrypted) constants used in the AES decryption routine, but there may be others that will work (does anybody know?). Corollary 2 prevents the operator constructing a program to emit any one of the tuples of encrypted constants that will do, with any probability above chance. Theorem 3 prevents the operator doing it without programmed help. □

## References


1. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU-based attestation and sealing. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013). ACM, New York (2013)
2. Biryukov, A.: Known plaintext attack. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, pp. 704–705. Springer, Boston (2011). https://doi.org/10.1007/978-1-4419-5906-5
3. Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Stam, M. (ed.) IMACC 2013. LNCS, vol. 8308, pp. 45–64. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45239-0_4
4. Breuer, P.T., Bowen, J.P.: A fully homomorphic crypto-processor design. In: Jürjens, J., Livshits, B., Scandariato, R. (eds.) ESSoS 2013. LNCS, vol. 7781, pp. 123–138. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36563-8_9
5. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: On obfuscating compilation for encrypted computing. In: Proceedings of the 14th International Conference Security and Cryptography (SECRYPT 2017), pp. 247–254. SCITEPRESS, Portugal, July 2017. https://doi.org/10.5220/0006394002470254
6. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: Superscalar encrypted RISC: the measure of a secret computer. In: Proceedings of the 17th International Conference on Trust, Security and Privacy in Computer and communications (TrustCom 2018), pp. 1336–1341. IEEE, CA, August 2018. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00184
7. Buer, M.: CMOS-based stateless hardware security module. U.S. Patent Application No. 11/159,669, April 2006. https://patents.google.com/patent/US20060072748
8. Conway, J.H.: FRACTRAN: a simple universal programming language for arithmetic. In: Cover, T.M., Gopinath, B. (eds.) Open Problems in Communication and Computation, pp. 4–26. Springer, Heidelberg (1987). https://doi.org/10.1007/978-1-4612-4808-8_2
9. Fletcher, C.W., van Dijk, M., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing (STC 2012), pp. 3–8. ACM, New York (2012). https://doi.org/10.1145/2382536.2382540
10. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC 2009), NY, pp. 169–178 (2009). https://doi.org/10.1145/1536414.1536440
11. Gentry, C., Halevi, S.: Implementing gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20465-4_9
12. 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.) CRYPTO 2013. LNCS, vol. 8042, pp. 75–92. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_5
13. Goldwasser, S., Micali, S.: Probabilistic encryption and how to play mental poker keeping secret all partial information. In: Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC 1982), pp. 365–377. ACM, New York (1982). https://doi.org/10.1145/800070.802212

14. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache attacks on Intel SGX. In: Proceedings of the 10th European Workshop on Systems Security (EuroSec 2017), pp. 2:1–2:6. ACM (2017). https://doi.org/10.1145/3065913.3065915
15. Hardin, D.: Real-time objects on the bare metal: an efficient hardware realization of the JavaTM virtual machine. In: Proceedings of tne 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), pp. 53–59. IEEE Computer Society, Washington (2001). https://doi.org/10.1109/ISORC.2001.922817
16. Hartman, R.: System for seamless processing of encrypted and non-encrypted data and instructions. US Patent 5,224,166, 29 June 1993. https://patents.google.com/patent/US5224166
17. Hashimoto, M., Teramoto, K., Saito, T., Shirakawa, K., Fujimoto, K.: Tamper resistant microprocessor. US Patent 2001/0018736 (2001). https://patents.google.com/patent/US20010018736A1
18. Irena, F., Murphy, D., Parameswaran, S.: CryptoBlaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support. In: Proceedings of the 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 702–708. IEEE (2018). https://doi.org/10.1109/ASPDAC.2018.8297404
19. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
20. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. arXiv:1801.01203 [cs.CR], January 2018. https://dblp.org/rec/bib/journals/corr/abs-1801-01203
21. Lipp, M., et al.: Meltdown. arXiv:1801.01207 [cs.CR], January 2018. https://dblp.org/rec/bib/journals/corr/abs-1801-01207
22. Ostrovsky, R., Goldreich, O.: Comprehensive software protection system. US Patent 5,123,045, 16 June 1992. https://patents.google.com/patent/US5123045
23. Patterson, D.A.: Reduced instruction set computers. Commun. ACM **28**(1), 8–21 (1985). https://doi.org/10.1145/2465.214917
24. Patterson, D.A., Hennessy, J.: Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, San Mateo (1994)
25. Rass, S., Schartner, P.: On the security of a universal cryptocomputer: the chosen instruction attack. IEEE Access **4**, 7874–7882 (2016). https://doi.org/10.1109/ACCESS.2016.2622724
26. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Found. Secure Comput. **4**(11), 169–180 (1978)
27. Sinha Roy, S., Järvinen, K., Vercauteren, F., Dimitrov, V., Verbauwhede, I.: Modular hardware architecture for somewhat homomorphic function evaluation. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 164–184. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48324-4_9
28. Schoeberl, M.: Java technology in an FPGA. In: Becker, J., Platzner, M., Vernalde, S. (eds.) FPL 2004. LNCS, vol. 3203, pp. 917–921. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30117-2_99
29. 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: Proceedings of IEEE Symposium on Security and Privacy, pp. 38–54, May 2015. https://doi.org/10.1109/SP.2015.10

30. Tsoutsos, N.G., Maniatakos, M.: The HEROIC framework: encrypted computation without shared keys. IEEE Trans. CAD IC Syst. **34**(6), 875–888 (2015). https://doi.org/10.1109/TCAD.2015.2419619
31. Wang, Z., Lee, R.B.: Covert and side channels due to processor architecture. In: Proceedings of the 2nd Annual Computer Security Applications Conference (ACSAC 2006), pp. 473–482. IEEE (2006). https://doi.org/10.1109/ACSAC.2006.20
32. Weicker, R.: Dhrystone: a synthetic systems programming benchmark. Commun. ACM **27**(10), 1013–1030 (1984). https://doi.org/10.1145/358274.358283