Encrypted Computing
Speed, Security and Provable Obfuscation against Insiders

Peter T. Breuer
Hecusys LLC
Atlanta, GA

Jonathan P. Bowen
London South Bank University
London, UK

Esther Palomar
Birmingham City University
Birmingham, UK

Zhiming Liu*,†
Southwest University
Chongqing, China

***Abstract*—Over the past few years we have articulated theory that describes 'encrypted computing', in which data remains in encrypted form while being worked on inside a processor, by virtue of a modified arithmetic. The last two years have seen research and development on a standards-compliant processor that shows that near-conventional speeds are attainable via this approach. Benchmark performance with the US AES-128 flagship encryption and a 1 GHz clock is now equivalent to a 433 MHz classic Pentium, and most block encryptions fit in AES's place.**

This summary article details how user data is protected by a system based on the processor from being read or interfered with by the computer operator, for those computing paradigms that entail trust in data-oriented computation in remote locations where it may be accessible to powerful and dishonest insiders. We combine: (i) the processor that runs encrypted; (ii) a slightly modified conventional machine code instruction set architecture with which security is achievable; (iii) an 'obfuscating' compiler that takes advantage of its possibilities, forming a three-point system that provably provides cryptographic 'semantic security' for user data against the operator and system insiders.

I. Introduction

This paper examines *encrypted computing*. That refers to a processor or other computing platform (a virtual machine, for example) that accepts encrypted inputs and produces encrypted outputs. The encrypted computing platform runs machine code programs in which the constants and possibly more are in encrypted form. A processor that supports encrypted computing in principle is suited as a platform for remote computations in the cloud [1] on behalf of a user who wants an assurance that an insider in the computer room is unable to access the data being processed. The user will encrypt the program and the input data on their own machine and send them across the network to the server, which executes it and produces the encrypted results. Those are sent back to the user who decrypts them on their own machine.

An analysis by van Dijk and Juels [2] shows that the goal of privacy and security of the user's data in this situation is equivalent to the condition that data is *cryptographically obfuscated* [3] from *the operator and operating system as adversary* on the encrypted computing platform.

Cryptographic obfuscation means that there is no deterministic or statistical method that gives the operator or any other adversary on the platform an advantage in deciphering the data input to or produced by the user's program at runtime. That is as compared with the success that the attack would have if it were applied to a black box that produced the program's (encrypted) outputs from its (encrypted) inputs by fiat, via absolutely no intermediate and/or internal computational states at all. Cryptographic obfuscation means that whatever the operator's privileges are on the encrypted computing platform – they conventionally include unrestricted access to all processor registers and all memory locations at all times – they are of no advantage in cracking the encryption. If obfuscation works, being able to see the code and how it runs on the machine from step to step and being able to experiment with changing code and/or data affords no leverage to the operator.[1]

So the question is if data can be cryptographically obfuscated from the operator and if the system for that is practical. The answer given here is 'yes'. This paper describes a largely conventional processor that 'works encrypted' at near normal speeds and in which the operator has all the ordinary privileges, plus a machine code, and a compiler, such that the three together provably cryptographically obfuscate user data from the operator. The complete system constitutes a platform for remote encrypted computing that maintains the privacy and security of user data against other users and the operator to the maximum extent possible.[2] Its conventional aspect means that known techniques may be applied to make it work even faster in the future than the prototype already does. A toolchain and minimal operating system is already in place.[3]

Although we do not seek to secure user data from physical probes, protection against the operator implies that some physical attacks, such as 'cold boot' (freezing the RAM sticks for later examination) [7], [8], [9] are prevented. The operator has access to RAM, so defending against the operator must

*Zhiming Liu wishes to thank the Chinese NSF for support from research grant 61672435, and Southwest University for research grant SWU116007.

†Correspondence: Zhiming Liu, RISE, 2 Tiansheng Rd, Beibei, 400715 China.

[1] Cryptographic obfuscation cannot always stop an adversary. Barak et al. [4] exhibited functions that cannot be disguised one as another even with obfuscation, and therefore claimed that obfuscation is impossible in general. But those functions $f(x)$ cannot be told from $f(x+A)+B$ in the context described here, for any constants A, B – so even if what x is intended to mean can be guessed, it is not known what number represents it under the encryption. That leaves 'wriggle room' for cryptographic obfuscation to work in the present context, despite Barak et al.'s famous result. It may be attributed to (i) inputs and outputs are in encrypted form here, and (ii) the hardware makes available only nonstandard primitive arithmetic and logical operations.

[2] Keys may be embedded at manufacture, as with Smart Card technologies [5] or introduced as needed via a Diffie-Hellman circuit [6] or equivalent that loads the key safely in public view. without revealing it to even the operator.

[3] There is no compromise from running with the wrong key in the machine. A program compiled with the right key does not work and a program that works' inputs and outputs are unwritable and unreadable.

978-1-5386-1585-0/17/$31.00

logically defend against cold-boot too and indeed, as that logic dictates, user data in RAM will always be encrypted.

This system is in the first instance aimed at remote *offline* ('batch') computation, because the same program must be recompiled by the 'obfuscating' compiler and reencrypted (but *not* with a different key) by the assembler whenever a new set of data is ready. However, if the future inputs are all known beforehand, a loop in the code and the one compilation suffice. However, we believe that continuous operation will soon be possible. Our prognostication is that the obfuscating scheme embedded by the compiler may be varied by the user while the program is running, with no change of encryption key.

The layout of this paper is as follows. Section II describes the 'homomorphic systems' that form the basis of our approach, and gives details and measures of our particular processor design for encrypted computing. Security problems that intrinsically arise from encrypted computing are considered in Section III. Section IV offers a partial solution via a restricted but computationally complete set of machine code instructions, and Section V offers a complete and practical solution via a 'fused arithmetic' style of instruction set. Section VI introduces an obfuscating compiler. Formal statements of security (mostly without formal proof here, but sketched or referenced) against deterministic attacks are given in sections IV and V, and against stochastic attacks in Section VI.

II. Homomorphic systems

A *homomorphic system* for encrypted computation is one for which *the inputs, outputs and all observable internal states are encrypted versions of what would be seen in a conventional processor running the same program.*

There is a simple canonical example: Ascend [10] is a prototype processor to which nobody, including the operator, has access except via its external input/output (I/O) pins. It expects encrypted data and machine code (Ascend's encryption is 128-bit AES [11]), and it returns encrypted data. Communication with memory is encrypted, via 'oblivious RAM' [12], [13], [14]. The statistics with respect to external cache hits, memory address patterns, power use patterns, etc, must be 'prophesied' prior to a run and the processor works to make it come true. There are no intermediate states that can be observed. Ascend is a coprocessor (i.e., a unit called upon by the main processor for special tasks) so it runs no interrupt handlers or other operating system code. It unstoppably runs to completion an encrypted program on the encrypted data delivered to it.

The platform is effectively a black box, satisfying the Hada cryptographic obfuscation definition's requirement, but it is awkward to set up for each use, it runs 12 to 13.5 times slower in encrypted mode[4] and the suitable programs for it are limited. We would like something faster and less handicapping.

An improvement is obtained in our own homomorphic system, the KPU ('Krypto Processor Unit') processor [15]. It

[4]The Ascend processor prototype's hardware base is not a sophisticated design so its absolute speed cannot meaningfully be compared with a contemporary off-the-shelf processor, but comparing its encrypted with unencrypted running speed is a proxy for the 'efficiency' of that mode of operation.

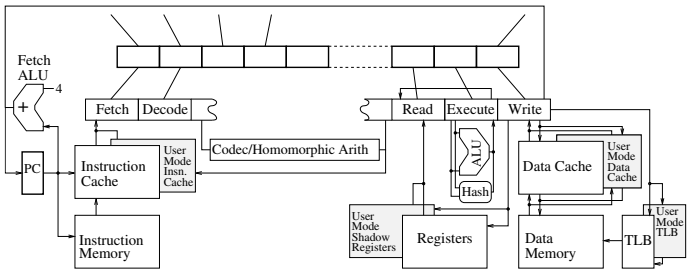

Fig. 1. Pipeline integration showing shadow units (shaded) for user mode.

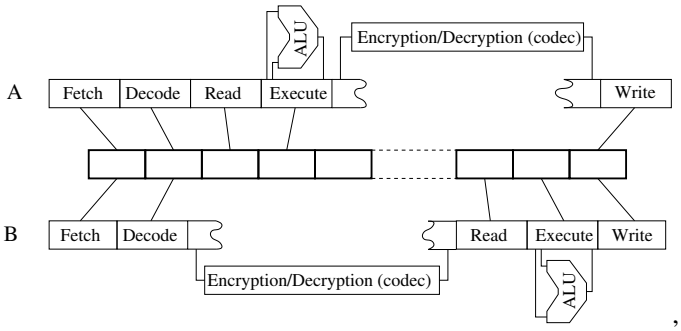

Fig. 2. The pipeline is configured in two different ways, 'A' and 'B', for different user mode instructions, in working with a symmetric encrypyion.

replaces the arithmetic unit in a conventional processor design with a non-standard *homomorphic arithmetic*.[5] It is shown in [16] that a processor working encrypted via that mechanism is a homomorphic system. Running encrypted, the processor's code and internal states are observable to all, but data is seen to be encrypted wherever it may be observed.

The architecture is detailed in [17]. It is a classic superscalar[6] design with a single pipeline [18]. It is modified to incorporate a non-standard arithmetic and has separate user mode and supervisor mode caches and other potential common information sources, as shown in Fig. 1. Embedding a symmetric encryption, the non-standard arithmetic arises through a conventional ALU working on unencrypted data in 'shadow' registers, where the first of a series of arithmetic operations triggers decryption to the shadow registers and the last engages encryption back again. The hardware protocol is proved secure in [19]. The pipeline is configured in two ways for two classes of instruction, as illustrated in Fig. 2. The 'A' configuration is for instructions that may use encryption only after arithmetic (the majority), the 'B' configuration is for those containing immediate data that need decryption first.

The block encryption fitted can be any that matches the machine word, RC2 (64 bit) and AES (128 bit) having been trialled.

[5]Let $\mathcal{E}$ be the encryption, then a homomorphic arithmetic $(f'_1, \ldots, f'_n)$ is one that *for each arithmetic operation f_i implements a corresponding operation f'_i that takes encrypted operands $[x]_\mathcal{E}$ and produces an encrypted output $[y]_\mathcal{E}$ where $y = f(x)$*. If there is some algebraic identity $e_1[\vec{f}](\vec{x}) = e_2[\vec{f}](\vec{x})$ in variables $\vec{x}$ for expressions e_i in operations $\vec{f}$ that holds of the original arithmetic, such as $x + y = y + x$, then $e_1[\vec{f'}]([\vec{x}]_\mathcal{E}) = e_2[\vec{f'}]([\vec{x}]_\mathcal{E})$ holds identically too, i.e., $[x]_\mathcal{E} +' [y]_\mathcal{E} = [y]_\mathcal{E} +' [x]_\mathcal{E}$ is true. The homomorphic arithmetic's algebra 'looks the same'.

[6]'Superscalar' means many instructions are worked on at once.

The data encryption is one-to-many, extra padding under the encryption varying pseudo-randomly. Encrypted addresses in particular can vary dynamically at runtime, producing 'hardware aliasing' from a software perspective, in which different memory locations are sporadically accessed by the same address. Programs must be compiled to cope with that [20].

The KPU runs the OpenRISC machine code described at opencores.org/or1k/Architecture_Specification, modified for encryption. The modifications principally extend some instructions to accommodate 64- or 128-bit encrypted constants. The KPU slows down by 10–40% in encrypted running under AES-128 as compared to running unencrypted (measures are from [17]), and with its base clock set at 1 GHz (3 ns cache), it runs like a 433 MHz classic Pentium,[7] so a KPU is quite fast.[8]

The KPU's nearest competitor among homomorphic systems is HEROIC [21], which has a stack machine architecture [22]. HEROIC embeds 2048-bit additively homomorphic Paillier encryption [23] in its encrypted working. Beneath the encryption, it is a 16-bit machine. An encrypted addition takes 4000 cycles on its 200 MHz base hardware, making HEROIC's speed roughly equivalent to a 25 KHz Pentium. A KPU is about 20,000 times faster. The point, however, is that the security of encrypted computations in these systems can be mathematically backed, as explained below.

III. Vulnerabilities of encrypted computation

Being able to run arbitrary computable functions encrypted is potentially dangerous to the security of an encryption. It is prima facie possible that an adversary may make some unforeseen use of the platform's computations to subvert the encryption. That is the danger that encrypted computing must guarantee to avoid, because the data is already in encrypted form and the security offered is as strong as the encryption is, barring the extra vulnerabilities that derive from computations.

There is real danger here, because in order not to confound the programmer, a platform for encrypted computing ought to implement the familiar computer arithmetic. That means 2s complement arithmetic, for example 32-bit, in which 32 additive doublings of anything obtains an encrypted zero. Using $[\cdot]_{\mathcal{E}}$ to represent encryption:

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

There will be on any conventionally programmable platform an instruction that takes an encrypted value $[x]_{\mathcal{E}}$ in one register and causes it to be replaced in the register by the encryption $[x+x]_{\mathcal{E}}$ of the same value added to itself. Then the encryption is open to a 'known plaintext attack' (KPA) against 0.

Similarly with a multiplication instruction: an adversary choosing a random encrypted value $[x]_{\mathcal{E}}$ has a 50% chance of picking an odd number x, and then repeated self-application of the multiplication instruction produces an encrypted 1, by Fermat's Little Theorem: $[x * \cdots * x]_{\mathcal{E}} = [x^{2^{31}} \mod 2^{32}]_{\mathcal{E}} = [1]_{\mathcal{E}}$.

A 1 may also be gotten if there is a division instruction, via $[x/x]_{\mathcal{E}} = [1]_{\mathcal{E}}$ if nonzero x is chosen, which is nearly certain.

Obtain an encrypted 1 by whatever means and then, by repeated addition, an adversary may first build all the $[2^k]_{\mathcal{E}}$ and then efficiently build the encryption $[K]_{\mathcal{E}}$ of any desired number K from its binary representation via

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

Conversely, given an encrypted 1 and an instruction for arithmetic comparison, any encrypted number $[x]_{\mathcal{E}}$ can be decrypted by comparing it with each constructed integer $[K]_{\mathcal{E}}$ in turn [24]. Or, more efficiently, by deducing its digits, comparing and subtracting 2^k from K when $2^k \leq x < 2^{k+1}$. That requires only an instruction for encrypted subtraction, in addition to those for encrypted addition and comparison.

In case there are no instructions for encrypted multiplication or division, a subroutine may do the job using only encrypted addition and comparison. The subroutines wil require an encrypted 1 as a constant in the code, and that may be repurposed as-is by an adversary who locates the subroutine.

So there is a case to answer with respect to security. Fortunately, the answer is positive.

IV. Secure encrypted computation

Despite the alarms above, there are ways of running arbitrary encrypted computations securely. Consider a minimal computationally complete subset of instructions[9] consisting of *addition of a constant* $y \leftarrow x+k$ (in human-readable form) and branches based on *comparison with a constant* $x<K$. Consider a program C *written using only those two instructions*. By a 'method of observation' understand a deterministic non-interfering process, based on observing what a running user program does from step to step – the *trace* T. Given *the operator cannot read the encryption*, then:

> **Fact 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.

The proof is given in [27]. The argument[10] assumes encryption

[7] Simulations show a Dhrystones v2.1 MIPS rating of 104-140 for the KPU with AES-128 and RC2-64 respectively, and 48.1 for a 200 MHz Pentium; see Dhrystones table at http://www.roylongbottom.org.uk/dhrystone results.htm.

[8] The authors are not Intel or AMD, and it may be supposed that experts such as those would do far better at implementation. In that respect, the KPU prototype's slowdown of 10-40% running encrypted indicates what may eventually be achieved relative to a PC too.

[9] A practitioner's 'proof' of the computational completeness of those two instructions is the mathematician J.H. Conway's well-known *Fractran* programming language [25], in which those are the only instructions. Attention in the computer hardware community may have been first drawn to the fact by Patterson and Hennessy in [26].

[10] The argument for Fact 1 supposes the hypothetical method has trace T and code C as inputs. Construct a modified code C' such that (i) it has a trace T' that 'looks the same' as T to the operator, because the differences are only in encrypted data, not control flow; (ii) the code C' 'looks the same' as C too (detail follows). Then the operator's method must give the same result applied to C', T' as it does applied to C, T, which is y, say. However, (iii) the new code C' gives the output $[y+7]_{\mathcal{E}}$ when run, not $[y]_{\mathcal{E}}$. So the method is wrong and does not work. The program C' differs from C only in the encrypted constants $[K]_{\mathcal{E}}$ in the $[x<K]_{\mathcal{E}}$ test in branch instructions. Those are changed to compensate for underlying data x now universally taking value 7 more than before. Changing K in C to $K'=K+7$ in C' achieves that. Nothing else changes from C. Then C' does exactly the same at runtime as C does, but on data that is everywhere $[x+7]_{\mathcal{E}}$ instead of the $[x]_{\mathcal{E}}$ when C runs.

is secure and the result is relative to the encryption strength.

Unfortunately programs limited like C are not typical in a KPU, running OpenRISC machine code. A counter-example to the result is provided by the one-instruction program consisting of 'sub[tract] r,r,r' (replace $[x]_{\mathcal{E}}$ in register r with $[x - x]_{\mathcal{E}}$). That produces $[0]_{\mathcal{E}}$, and any onlooker can deduce that by the singularly deterministic method of reading the instruction while knowing the laws of arithmetic. The problem is the instruction set, and it will be mended in Section V.

Now consider program C again. Set *the constants* $[K]_{\mathcal{E}}$, $[k]_{\mathcal{E}}$ *in the program to come from disjoint subspaces of the cipherspace, themselves disjoint from data circulating in the processor*. The easy way to arrange that in a KPU processor is to incorporate distinct markers into the padding under a symmetric encryption. Then:

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

The reason is that the program hypothetically built by the operator is the kind of program forbidden to exist by Fact 1, for which the output $[y]_{\mathcal{E}}$ is known to be an encryption of y. It does not matter who has written the program (this reasoning obviates a repeat here of the argument[10] for Fact 1).

That answers the natural suspicion that an attacker could turn some mechanism of the processor to the task of decrypting program data. The answer is 'no', if the instructions are just add-a-constant and branch on compare-to-a-constant. The answer is also 'no' if the attacker is prevented from running their own programs on user data, as in Ascend. The conclusion does not apply to KPUs, which support an extensive conventional instruction set, and the fix is addressed in the rest of this paper.

V. FxA instruction architecture

The KPU, which runs the whole of the standard 32-bit integer and floating point OpenRISC instruction set (there are approximately 200 different instructions) in encrypted mode, is vulnerable to attacks on the encryption via the example in Section IV on the program consisting of one subtraction instruction with two arguments that can be used to generate an encrypted zero. However, there are also many of its instructions that a KPU may safely execute. Analysis shows that binary addition, binary subtraction, subtraction of a constant, multiplications, left shifts (not division, right shift) are all safe to execute, *provided* each instruction is atomically followed by addition of some constant. That could be enforced in the processor, but there is opportunity here to co-opt instead a single new *fused multiply and add* (FMA) instruction[11] to cover the instruction combinations mentioned.

[11]Fused multiply and add comes about because, while addition takes one cycle to complete on most processors, multiplication takes much longer (typically ten cycles). The logic subunit that is repeated to form a multiplication multiplies two short integers plus two short carries from subunits 'right' and 'below' in a 2-dimensional array. The column and row of subunits at extreme 'right' and 'bottom' usually has the carry inputs tied to zero, but they may be used to feed two extra full integer addends into the calculation at no cost, producing $x * y + z + w$ instead of just $x * y$. That is the FMA intruction.

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, for the 'FMA3/4' instruction sets. Denote by a *fused anything and add* (FxA) architecture one in which no arithmetic instruction appears except as the fused compound with addition of a constant on inputs and outputs. So FxA multiplication does the following under the encryption:

$$x_3 \leftarrow (x_1 - k_1) * (x_2 - k_2) + k_3$$

where the k_i, i=1, 2, 3 are constants embedded encrypted in the instruction. Some instructions, such as binary addition, need only *one* constant incorporated in the instruction, as

$$(x_1 - k_1) + (x_2 - k_2) + k_3 = x_1 + x_2 + (k_3 - k_1 - k_2)$$

Section IV requires the processor to enforce *no collisions between (i) encrypted constants embedded in instructions and (ii) runtime encrypted data values circulating in registers or memory*. With that, the arguments for Facts 1 and 2 can be applied with an FxA-conformant instruction set too[12]. Again, by 'method' understand a deterministic procedure. Then:

Fact 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.*

An extra decode stage on the front lets the KPU translate a FxA-compliant instruction set to OpenRISC, letting Fact 3 apply. But deterministic methods of attack are not the only danger. An attack may fail almost always and still succeed enough times to make it worthwhile trying, so stochastic considerations need to be taken into account, as below.

VI. Obfuscating compilation

To make compilation to an encrypted FxA-compliant instruction set vary stochastically in a way that is cryptographically significant at runtime, the compiler sets a different *offset* $\mathbf{d}x_l$ from nominal for the value $[x_l + \mathbf{d}x_l]_{\mathcal{E}}$ in each register and memory location l, at different points in the program. At each recompilation of the same source, the compiler will choose different offsets, randomly, with uniform distribution. In consequence, runtime values will vary, with uniform distribution, from compilation to compilation. The machine code produced by the compiler will look the same, up to differences in the embedded encrypted constants, which the adversary cannot decrypt. The traces will also look the same.

Consider a boolean expression $A \,\&\&\, B$ in the source code. The compiler will

(a) already know if A is compiled telling the truth or lying;
(b) already know if B is compiled telling the truth or lying;

[12]In a given program C, every FxA instruction can be changed via adjustments in its embedded (encrypted) constants to accommodate every data value passing through registers and memory to be 7 more than it used to be under the encryption, for example, as argued in footnote[10] in support of Fact 1, which implies Fact 2.

(c) decide randomly if it will lie or tell truth for $C{=}A\&\&B$.

The (a) corresponds to whether $\mathbf{d}$a=0 (truth) or $\mathbf{d}$a=1 (liar) is deliberately offset by the compiler for the nominal runtime value of the one-bit integer representing A. Similarly (b) corresponds to whether $\mathbf{d}$b=0 (truth) or $\mathbf{d}$b=1 (liar) is offset by the compiler for B. Let a be a symbolic expression that is true when (a) is set to tell truth ($\mathbf{d}$a=0), and false when (a) is set to lie ($\mathbf{d}$a=1). Similarly for b with respect to (b), and c with respect to (c). Then what is computed at runtime is

$$c \leftrightarrow ((a \leftrightarrow A)\&\&(b \leftrightarrow B))$$

where the two-sided arrow stands for the boolean biconditional operator (the complement of exclusive or). That is

if $a\,b\,c$ then $A\&\&B$	if $a\,b\,\overline{c}$ then $\overline{A\&\&B}$
if $\overline{a}\,b\,c$ then $\overline{A}\&\&B$	if $\overline{a}\,b\,\overline{c}$ then $\overline{\overline{A}\&\&B}$
if $a\,\overline{b}\,c$ then $A\&\&\overline{B}$	if $a\,\overline{b}\,\overline{c}$ then $\overline{A\&\&\overline{B}}$
if $\overline{a}\,\overline{b}\,c$ then $\overline{A}\&\&\overline{B}$	if $\overline{a}\,\overline{b}\,\overline{c}$ then $\overline{\overline{A}\&\&\overline{B}}$

where the overline means boolean negation. The compiler knows a and b and generates c with 50/50 probability, so deciding which of $A\&\&B$, $\overline{A}\&\&B$, etc., it will generate machine code for. All the generated codes look the same, modulo encrypted constants, which are unreadable by the operator-as-adversary. The sequence has the classical form that a compiler should emit for $A\&\&B$. In particular, the branch takes the 'short circuit' route to an early out if A fails.

With other source code constructs, the compiler[13] randomly generates differences from nominal that run the full 32-bit range, not the 1 bit of the boolean case. That claim has been tested by compiling the code for the Ackermann function[14] [28] and running it with (3,1) as input. The program gives result $[13]_{\mathcal{E}}$ in 8922 instructions executed. Table I shows the object code with its randomly generated embedded constants. For clarity in the trace (shown in Table II), the offsets $\mathbf{d}y$ for the return value from functions and $\mathbf{d}x$ for function parameters have been set at 0 in the code, instead of being random (entry and exit offsets for a complete executable program would be made known to the remote user), allowing the trace to finish with the perfect result in register **v0**, which is OpenRISC's specified application binary interface return value register.

In [27] it is proved of this compile strategy that:

Fact 4. *The probability across different compilations that any particular 32-bit value x has its encryption $[x]_{\mathcal{E}}$ in a given register or memory location at any given point in the program at runtime is uniformly $1/2^{32}$.*

[13]Haskell code for the compiler and a virtual machine to run the object code may be downloaded from http://nbd.it.uc3m.es/~ptb/obfusc_comp-0_7.hs.

[14]The C source code for the Ackermann function is as follows:

```
int A(int m, int n) {
  if (m <= 0) return n+1;
  if (n <= 0) return A(m-1, 1);
  return A(m-1, A(m, n-1));
};
```

An increment in the first argument produces an exponential increment with respect to the second argument, $A(m,n) = 2^{\cdots 2^{n+3}} - 3$. The ellipsis covers $m-2$ exponentiations. The first exponential case is $A(3,n) = 2^{n+3} - 3$.

TABLE I
FXA ENCRYPTED MACHINE CODE FOR THE ACKERMANN FUNCTION

```
0   A:  ...                  # create frame
...                          # if (m == 0)
4   add  t0  a0  zer [-1704185953]ε    # m
5   add  t1  zer zer [2104023132]ε     # 0
6   sfeq t0  t1      [486758211]ε      # ==
...
15  bnf 9                      # then
                               # return n + 1
16  add  t0  a1  zer [1526256091]ε     # n
17  add  t1  zer zer [1280102991]ε     # 1
18  add  t0  t0  t1  [-620124265]ε     # +
19  add  v0  t0  zer [2108732479]ε
...                          # frame destroy
...                          # return sequence
24  b 0                        # else skip
                               # if (n == 0)
25  add  t0  a1  zer [-989123886]ε     # n
26  add  t1  zer zer [-580299623]ε     # 0
27  sfeq t0  t1      [-408824263]ε     # ==
...
36  bnf 26                     # then
                               # return  A(m-1,1)
37  add  t0  a0  zer [-457757118]ε     # m
38  add  t1  zer zer [-587705998]ε     # 1
39  sub  t0  t0  t1  [-2141902894]ε    # -
40  add  t1  zer zer [1111468055]ε     # 1
...                          # save regs
...                          # fill args
50  jal 0                      # call  A(m-1,1)
51  add  t0  v0  zer [-1607308215]ε
...                          # restore regs
57  add  v0  t0  zer [1607308215]ε
...                          # destroy frame
...                          # return sequence
62  b 0                        # else skip
                               # return A(m-1,A(m,n-1))
63  add  t0  a0  zer [1195221673]ε     # m
64  add  t1  zer zer [-868884270]ε     # 1
65  sub  t0  t0  t1  [-1733760489]ε    # -
66  add  t1  a0  zer [-1996082249]ε    # m
67  add  t2  a1  zer [-1268351424]ε    # n
68  add  t3  zer zer [1148618604]ε     # 1
69  sub  t2  t2  t3  [752318999]ε      # -
...                          # save regs
...                          # fill args
79  jal 0                      # call A(m,n-1)
80  add  t1  v0  zer [-191727838]ε
...                          # restore regs
...                          # save regs
...                          # fill args
95  jal 0                      # call A(m-1,..)
96  add  t0  v0  zer [1566613208]ε
...                          # restore regs
102 add  v0  t0  zer [-1566613208]ε
...                          # destroy frame
...                          # return sequence
```

Machine instructions:

opcode	*fields*	*semantics*	
add	$r_0\,r_1\,r_2\,[k]_{\mathcal{E}}$	add	$r_0 \leftarrow [[r_1]_{\mathcal{D}} + [r_2]_{\mathcal{D}} + k]_{\mathcal{E}}$
sub	$r_0\,r_1\,r_2\,[k]_{\mathcal{E}}$	subtract	$r_0 \leftarrow [[r_1]_{\mathcal{D}} - [r_2]_{\mathcal{D}} + k]_{\mathcal{E}}$
sfeq	$r_1\,r_2\,[k]_{\mathcal{E}}$	set flag if $[r_1]_{\mathcal{D}} = [r_2]_{\mathcal{D}} + k$ else clear it	
bf	j	skip j instructions if flag set else continue	
bnf	j	skip j instructions if flag unset else continue	
b	j	unconditional skip j instructions	
jal	l	jump to address l, save return address in **ra** register	

The r are register indices or memory locations, the k are 32-bit integers, the j are instruction address increments, '$\leftarrow$' is assignment. The function $[\,\cdot\,]_{\mathcal{E}}$ represents encryption, $[\,\cdot\,]_{\mathcal{D}}$ represents decryption of a value or register/memory content.

It is argued in [27] that that means the runtime data is cryptographically *semantically secure* [3] against the operator, provided the embedded (encrypted) constants in instructions cannot be deciphered, under the assumption of Sections IV, V

TABLE II
RUNTIME TRACE FOR THE ACKERMANN FUNCTION ON (3,1), RESULT 13.

```
PC      instruction                          update
...
4       add  t0  a0  zer [-1704185953]ε  t0 = [-1704185951]ε
5       add  t1  zer zer [2104023132]ε   t1 = [2104023132]ε
6       sfeq t0  t1      [486758211]ε
7       bf 2
8       sflt zer zer     [-520344919]ε
9       b 1
11      bf 2
12      sflt zer zer     [-686785144]ε
13      b 1
15      bnf 9
25      add  t0  a1  zer [-989123886]ε   t0 = [-989123885]ε
26      add  t1  zer zer [-580299623]ε   t1 = [-580299623]ε
27      sfeq t0  t1      [-408824263]ε
28      bf 2
...
102     add  v0  t0  zer [-1566613200]ε  v0 = [13]ε
...
106     jr ra
STOP
```

that *collisions between encrypted program constants and (encrypted) runtime data are impossible*.

VII. FUTURE WORK

It is planned to install an extra decode stage at the front of the KPU pipeline to support FxA instructions, now the theory of how instruction set design may provide security is understood.

The obfuscating C compiler presently only works with integer types, arrays being treated as long integers. 'Upgrading' to allow general pointers is planned, to complete the type cover. But a given pointer must always point into its designated area, which requires strengthening the C compiler's type discipline.

VIII. CONCLUSION

This paper has considered the security of encrypted computation on behalf of a remote user against the system operator or operating system as adversary. It has presented a fast prototype processor that, in user mode, works 'homomorphically' with respect to a conventional processor producing encrypted outputs from encrypted inputs via encrypted internal states. A 'FxA' machine code instruction set for the processor has been described that allows code and runtime (encrypted) data on the platform to be conventionally read and write accessible for the operator while privacy for the user's data is obtained. In conjunction with obfuscating compilation as described here, the decrypted data read and written by user programs provably cannot be deduced or even statistically estimated to any degree above chance. Our prototype processor runs encrypted at about the speed of a 500 MHz classic Pentium.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[2] M. van Dijk and A. Juels, "On the impossibility of cryptography alone for privacy-preserving cloud computing." *HotSec*, vol. 10, pp. 1–8, 2010.

[3] S. Hada, "Zero-knowledge and code obfuscation," in *Proc. 6th Int. Conf. Theory and Applicat. Cryptol. and Inform. Sec. (ASIACRYPT'00)*, ser. LNCS, T. Okamoto, Ed. Springer, 2000, no. 1976, pp. 443–457.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Proc. 21st Ann. Int. Crypto. Conf.*, ser. LNCS, J. Kilian, Ed., no. 2139. Springer, Aug. 2001, pp. 1–18.

[5] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Proc. USENIX Workshop Smartcard Tech.* Berkeley, CA: USENIX, May 1999, pp. 9–20.

[6] M. Buer, "CMOS-based stateless hardware security module," Apr. 6 2006, US Pat. App. 11/159,669.

[7] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[8] P. Simmons, "Security through amnesia: A software-based solution to the cold boot attack on disk encryption," in *Proc. 27th Ann. Comp. Sec. Applicat. Conf. (ACSAC'11)*. New York: ACM, 2011, pp. 73–82.

[9] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *Proc. 8th Int. Conf. Avl. Rel. Sec. (ARES'13)*. IEEE, 2013, pp. 390–397.

[10] C. W. Fletcher, M. van Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proc. 7th Worksh. Scal. Trust. Comp. (STC'12)*. ACM, 2012, pp. 3–8.

[11] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer, 2002.

[12] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *Proc. 22nd Ann. Symp. Theor. Comput.*, ACM. ACM, 1990, pp. 514–523.

[13] R. Ostrovsky and O. Goldreich, "Comprehensive software protection system," Jun. 16 1992, US Patent 5,123,045.

[14] S. Lu and R. Ostrovsky, "Distributed oblivious RAM for secure two-party computation," in *Proc. 10th Theor. Crypto. Conf. (TCC'13)*, ser. LNCS, A. Sahai, Ed. Springer, Mar. 2013, no. 7785, pp. 377–396.

[15] P. T. Breuer and J. P. Bowen, "Towards a working fully homomorphic crypto-processor: Practice and the secret computer," in *Proc. Int. Symp. Eng. Sec. Softw. Syst. (ESSoS'14)*, ser. LNCS, J. Jörjens, F. Pressens, and N. Bielova, Eds. Springer, Feb. 2014, no. 8364, pp. 131–140.

[16] ——, "A Fully Homomorphic Crypto-Processor Design: Correctness of a Secret Computer," in *Proc. Int. Symp. Eng. Sec. Softw. Syst. (ESSoS'13)*, ser. LNCS, no. 7781. Springer, Feb. 2013, pp. 123–138.

[17] ——, "A Fully Encrypted Microprocessor: The Secret Computer is Nearly Here," *Procedia Comp. Sci.*, vol. 83, pp. 1282–1287, Apr. 2016.

[18] D. A. Patterson, "Reduced instruction set computers," *Commun. ACM*, vol. 28, no. 1, pp. 8–21, Jan. 1985.

[19] P. T. Breuer, J. P. Bowen, E. Palomar, and Z. Liu, "A Practical Encrypted Microprocessor," in *Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16)*, C. Callegari, M. van Sinderen, P. Sarigiannidis, P. Samarati, E. Cabello, P. Lorenz, and M. S. Obaidat, Eds. SCITEPRESS, 2016, pp. 239–250.

[20] P. T. Breuer and J. P. Bowen, "Avoiding hardware aliasing: Verifying RISC machine and assembly code for encrypted computing," in *Proc. 25th IEEE Int. Symp. Softw. Rel. Eng. Worksh. (ISSREW'14)*. IEEE, Nov. 2014, pp. 365–370.

[21] N. G. Tsoutsos and M. Maniatakos, "The HEROIC framework: Encrypted computation without shared keys," *IEEE Trans. CAD Integ. Circ. Syst.*, vol. 34, no. 6, pp. 875–888, 2015.

[22] D. Hardin, "Real-time objects on the bare metal: An efficient hardware realization of the JavaTM virtual machine," in *Proc. 4th IEEE Int. Symp. OO. Real-Time Dist. Comp. (ISORC'01)*. IEEE, 2001, pp. 53–59.

[23] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Theor. Applic. Crypto. Tech. (EUROCRYPT'99)*, ser. LNCS, no. 1592. Springer, Apr. 1999, pp. 223–238.

[24] S. Rass and P. Schartner, "On the security of a universal cryptocomputer: The chosen instruction attack," *IEEE Access*, vol.4, pp. 7874–7882, 2016.

[25] J. H. Conway, "Fractran: A simple universal programming language for arithmetic," in *Open Problems in Communications and Computation*, T. M. Cover and B. Gopinath, Eds. Springer, 1987, pp. 4–26.

[26] D. A. Patterson and J. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann, 1994.

[27] P. T. Breuer, J. P. Bowen, E. Palomar, and Z. Liu, "On obfuscating compilation for encrypted computing," in *Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17)*, P. Samarati, M. S. Obaidat, and E. Cabello, Eds. Portugal: SCITEPRESS, Jul. 2017, pp. 247–254.

[28] Y. Sundblad, "The Ackermann function, a theoretical, computational, and formula manipulative study," *BIT Num. Math*, vol.11, no.1, pp. 107–119, Mar. 1971.