# An Obfuscating Compiler

Peter T. Breuer[1]

Hecusys LLC, Atlanta, GA, ptb@hecusys.com

**Abstract.** Privacy for arbitrary encrypted remote computation in the cloud depends on the running code on the server being obfuscated from the standpoint of the operator in the computer room. This paper shows that that may be arranged on a platform with the appropriate machine code architecture, given the obfuscating compiler described.

## 1 Introduction

A famous result of van Dijk et al. in the security arena reduces security for arbitrary computations in the cloud to *obfuscation* of the machine code on the server [9]. 'Obfuscation' means that access to the code running on the server gives no advantage, logical or statistical, to the hypothetical spy in the computer room [11]. The computation must take encrypted inputs and produce encrypted outputs or the I/O could be read directly, but that can be arranged with the appropriate processor hardware. The argument of van Dijk et al. is that the remote user's program and the input for it could together be supplied as (encrypted) input to a virtual machine running on the server. Running the program securely in that environment means that the (encrypted) output must be returned as though from an impenetrable black box, and that is exactly what the – perfectly accessible – code of the virtual machine being 'obfuscated' means.

There are processor contexts that effectively produce this kind of obfuscation. An example is the Ascend co-processor [10], which executes code in 'Fort Knox'-style physical isolation, with no operator access to it while it is running. The external observables, such as power consumption, timing of I/O (including memory accesses), cache hit/miss ratio, etc., that might leak information via side-channels [18, 19], obey statistics that are configured beforehand and have no correlation with the running program, apart from the run time. The machine code executable is also input in encrypted form. Since the operator's access to the running program is physically restricted to no more than would be the case for a black box implementation, the program is obfuscated by definition.

However, a classic result of Barak et al. affirms that obfuscation of programs is generally *impossible* [2]. Reconciling that with Ascend's evident physically-based obfuscation implies that assumptions about what the operator can see of the code while running, or what form the code may take, may not have universal relevance. It turns out that one has to go only a little way along the path of Ascend in securing the processor from the operator, yet obfuscation of the code on the server is technically and practically achievable while still allowing that:

*The operator has the conventional full access to running code.*

That means access (via a debug exception) to registers and memory, while the user's code may be single stepped, repeated, retried, altered, etc. The 'magic' is in an appropriate machine code instruction architecture, as set out in Section 2.

However, software toolchain support is needed, otherwise the code itself might obviate all protections. For example, it might include a trojan library function that takes one hour to complete if the answer is (encrypted) '1', and two hours if the answer is (encrypted) '2', which slowly generates a codebook that translates the encryption for the interested observer[1]. Moreover, a human author will write code incorporating small numbers that can be guessed in a dictionary attack. A kind of *obfuscating compiler* is still required, and this paper provides it.

This compiler at each recompilation of the same program produces an executable that in the code and at any point in the runtime trace has arbitrarily and uniformly distributed differences in the data values (underneath the encryption) with respect to any other compilation. Otherwise, the code and trace looks the same, in particular the control flow visible in the trace.

The idea is that a spy does not know which compilation is being run. Suppose hypothetically that the malfeasant operator has a method of determining what the encrypted output or trace data is under the encryption at any point. Then, since that in fact is something different under the encryption for every possible compiled executable, and the operator's method must see no difference between the different compilations or traces because the differences that do exist are all in the (encrypted) constants and data that cannot be read by the operator, so the putative method cannot work (Theorem 1, Section 2).

For the argument to go through, there are two hardware requirements:

(1) Individual machine code instructions must be obfuscated.

That means that individual instructions are treated as Ascend treats an entire program, so they are individually as secure as an entire program is in Ascend. But, as in Ascend, the 'I/O' from the instruction must also be in encrypted form or it would be readable by the operator:

(2) Each machine code instruction must read and write data in encrypted form.

otherwise the operator, able to single-step the machine and with access to registers, can read every action. However, (1) and (2) are not necessarily without overlap, as obfuscation and encryption are equivalent in some contexts [1].

There are platforms that satisfy those requirements. HEROIC [17, 16] is a prototype 16-bit machine running with a Paillier (2048 bit) encryption [14]. Its core performs a single machine code addition instruction in 4000 cycles and

[1] A side-channel consisting of signalling via repeat accesses to the same memory location is also closed in Ascend by the use of *oblivious RAM* [13], which remaps the logical to physical address relation dynamically, maintaining aliases, so access patterns are statistically impossible to spot. It also masks programmed accesses in a sea of independently generated extra random accesses.

20us on its 200MHz (programmable) hardware, roughly equal to the speed of a 25KHz Pentium. The addition instruction does addition on encrypted data, producing encrypted data, satisfying requirement (2). The 'OI' in HEROIC really stands for 'one instruction', meaning that the machine code architecture has only one kind of instruction, a combined 'add, compare, branch' that is known to be computationally complete together with recursion and at least one nonzero constant (the instruction by itself makes up Conway's Fractran programming language [7], often used in the mathematics of computability and complexity). Requirement (1) is satisfied because one instruction looks like another but for the (encrypted) constants embedded in it, which cannot be read.

The KPU design [6] generalises HEROIC's approach, achieving encrypted running by modifying the arithmetic logic unit (ALU) in a conventional processor design, which is enough in itself to cause data to be processed in encrypted form [3]. It does not require an encryption with special properties – the Paillier encryption in HEROIC has 'homomorphic' properties, in that addition under the encryption can be effected via multiplication of the encrypted values. The KPU runs in simulation using the US Advanced Encryption Standard (AES) 128 bit encryption [8] at about the speed of a 300-500MHz Pentium using a 1GHz clock [5], which is broadly comparable with contemporary PC processors.

The significant difference with respect to HEROIC, aside from speed, is that the machine code architecture is conventional, containing many instructions that are distinguishable to an observer who can read only the unencrypted parts. Nevertheless, it satisfies (2) as well as (1), as HEROIC does. The reason is that what certain instructions do to data is not knowable. Each of those instructions contains an encrypted parameter that is set by the compiler. It would be known to an observer what the intended result 0 of a 'traditional' machine instruction to calculate $x-x$ is (zero) even without being able to read the encrypted $x$, but it is not known of the KPU instruction that calculates $x-x+k$, because the parameter $k$ is encrypted. It can be proved that all outcomes are equally probable in these instructions when the parameter is uniformly distributed across recompilations.

Given (1) and (2) it will be shown here (Theorem 2, Section 4) that machine code is formally obfuscated by the simple 'obfuscating compiler' introduced in Section 3 and defined in Section 4 for platforms satisfying those requirements (Theorem 3), thus ensuring privacy for cloud computations for the compiled executables, via the argument of van Dijk in [9]. Section 5 gives an example of obfuscating compilation (and running the obfuscated code).

The compiler constructs machine code programs which cannot be distinguished one from the other, neither by look nor behaviour, by dissection nor experiment, by a spy ignorant of the encryption used for the constants in the code and the circulating data in the machine. Not all constructed programs look alike, but enough do to permit any interpretation of the inputs and outputs.

## 2 Encrypted FxA instruction architecture

The situation is clarified by a particular machine code instruction set. Instruction sets are conventionally made up of instructions that (a) perform a relatively simple *binary arithmetic operation*, such as 'addition' on data in registers or in memory and return a result to registers or memory, instructions that (b) perform a *binary comparison operation* such as 'less than' between two values in registers or memory, setting or clearing a flag, plus (c) *control instructions* that alter which program instruction is executed next. By default that is the one with the next higher address in memory, but a jump instruction unconditionally alters the sequence. A branch instruction alters the sequence conditionally on the value of the flag set by a previous comparison.

However, AMD and Intel in 2011-2013 introduced instruction sets that contain instructions (aa) that carry out a combination of *two* arithmetic operations at once, in a so-called *fused* instruction. They were introduced in the context of the newer processors such as Intel's 'Knights Landing' Xeon Phi that contain many (72) cores on the one chip, each running with very wide integers (512 bits).

While addition takes one cycle to complete on such processors, multiplication takes much longer (about ten cycles). Moreover, the repeating subunit that forms the multiplication logic multiplies two short integers and adds in two short incoming 'carry' integers from subunits 'right' and 'below' in a 2-dimensional array. The column and row of subunits at extreme 'right' and 'bottom' respectively may be used to feed two full integer addends into the calculation at no extra cost. Thus a 'fused multiply and add' (FMA) instruction was introduced in AMD and Intel's FMA3 and FMA4 instruction sets for reasons of efficiency. Compilers emit FMA instructions instead of single multiplications followed by add, particularly in connection with parallel matrix- and tensor-oriented computation.

Denote by a *fused anything and add* (FxA) instruction architecture one in which the arithmetic instructions subtract constants $k_1$, $k_2$ from the operands $x_1$, $x_2$ and add a constant $k_3$ into the result. So FxA multiplication does:

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

A concrete example of an FxA instruction set dealing with encrypted data is illustrated in Table 1. Some instructions, such as the addition instruction, for example, need only admit *one* constant addend, as

$$(x_1 - k_1) + (x_2 - k_2) + k_3 = x_1 + x_2 + k \text{ where } k = k_3 - k_1 - k_2$$

The constants (the $k$ in the above) appear in encrypted form in the instruction, so cannot be read by the operator, who does not have the encryption key. The instructions manipulate encrypted data, either, as in Ascend, by decrypting input and encrypting output, or, as in HEROIC, by making use of an encryption $\mathcal{E}$ with homomorphic properties such that the arithmetic may be carried out on the encrypted data as is. It only matters that the instructions are atomic.

For FxA instructions to work securely, the hardware should also ensure:

**Table 1.** An FxA machine code instruction set for working with encrypted data

| | *fields* | *kind* | *semantics* |
|---|---|---|---|
| add | $r_0\, r_1\, r_2\, [k]_\mathcal{E}$ | (aa) add | $r_0 \leftarrow [[r_1]_\mathcal{D} + [r_2]_\mathcal{D} + k]_\mathcal{E}$ |
| sub | $r_0\, r_1\, r_2\, [k]_\mathcal{E}$ | (aa) subtract | $r_0 \leftarrow [[r_1]_\mathcal{D} - [r_2]_\mathcal{D} + k]_\mathcal{E}$ |
| mul | $r_0\, r_1\, r_2\, [k_0]_\mathcal{E}\, [k_1]_\mathcal{E}\, [k_2]_\mathcal{E}$ | (aa) multiply | $r_0 \leftarrow [([r_1]_\mathcal{D} - k_1) * ([r_2]_\mathcal{D} - k_2) + k_0]_\mathcal{E}$ |
| div | $r_0\, r_1\, r_2\, [k_0]_\mathcal{E}\, [k_1]_\mathcal{E}\, [k_2]_\mathcal{E}$ | (aa) divide | $r_0 \leftarrow [([r_1]_\mathcal{D} - k_1) / ([r_2]_\mathcal{D} - k_2) + k_0]_\mathcal{E}$ |
| cmov | $r_0\, r_1\, r_2$ | (a) | conditional move $r_0 \leftarrow \text{flag}\,?\, r_2 : r_3$ |
| sfeq | $r_1\, r_2\, [k]_\mathcal{E}$ | (b) | set flag if $[r_1]_\mathcal{D} = [r_2]_\mathcal{D} + k$ else clear it |
| sfne | $r_1\, r_2\, [k]_\mathcal{E}$ | (b) | set flag if $[r_1]_\mathcal{D} \neq [r_2]_\mathcal{D} + k$ " |
| sflt | $r_1\, r_2\, [k]_\mathcal{E}$ | (b) | set flag if $[r_1]_\mathcal{D} < [r_2]_\mathcal{D} + k$ " |
| sfgt | $r_1\, r_2\, [k]_\mathcal{E}$ | (b) | set flag if $[r_1]_\mathcal{D} > [r_2]_\mathcal{D} + k$ " |
| sfle | $r_1\, r_2\, [k]_\mathcal{E}$ | (b) | set flag if $[r_1]_\mathcal{D} \leq [r_2]_\mathcal{D} + k$ " |
| sfge | $r_1\, r_2\, [k]_\mathcal{E}$ | (b) | set flag if $[r_1]_\mathcal{D} \geq [r_2]_\mathcal{D} + k$ " |
| bf | $j$ | (c) | skip $j$ instructions if flag set else continue |
| bnf | $j$ | (c) | skip $j$ instructions if flag not set else continue |
| b | $j$ | (c) | unconditional skip $j$ instructions |
| ... | | | |

Legend: the $r$ are register indexes 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. Kind (a) are arithmetic, (aa) fused arithmetic, (b) comparators, (c) control.

(3) There are *no collisions* between (i) encrypted constants $[k]_\mathcal{E}$ that appear in instructions and (ii) runtime encrypted data values in registers or memory.

In practice that is done by introducing different padding or blinding factors into the encryptions for the two, and checking that in the processor pipeline.

The idea is that the hardware should not allow constants that appear in instructions to work correctly when used as data inputs for arithmetic, and data generated at runtime should not work correctly as constants in instructions. That makes it impossible for a 'spy in the computer room' to pass the encrypted constants seen in other programs through the processor arithmetic, patching the results back into code snippets by way of experiment or tampering.

**Theorem 1.** *There is no method by which the privileged operator can read a program $C$ constructed using FxA instructions, nor deliberately alter it using those instructions to give an intended encrypted output.*

*Proof.* Suppose for contradiction that the operator has a method $f(T, C) = y$ of knowing that the output $[y]_\mathcal{E}$ of $C$ encrypts $y$, having observed the trace $T$. Now imagine that every number has 7 added to it under the encryption. Replace every FxA instruction of the form $r_0 \leftarrow (r_1 - k_1)\, \Theta (r_2 - k_2) + k_0$ with $r_0 \leftarrow (r_1 - k_1')\, \Theta (r_2 - k_2') + k_0'$ where $k_i' = k_i + 7$, $i = 0, 1, 2$. Then the operations still make sense, operating on numbers that are 7 more than they used to be to produce a number that is 7 more than it used to be. The comparisons $r_1\, R\, r_2 + k$ in C still make sense, because they compare numbers that are 7 more than they

used to be on both sides of the relation $R$, obtaining the same result, so they do not need to be changed. To the operator, the new program code $C'$ 'looks the same', $C' \sim C$, because one encrypted number $[k]_{\mathcal{E}}$ or $[k']_{\mathcal{E}}$ is as meaningful as another without the key and the new program trace $T'$ looks the same too up to the encrypted numbers in it, because the branches after comparisons go the same way since the comparisons still set the flag the same way. That is, $T' \sim T$. The method $f$ carried out by the operator must therefore declare the output of $C$ to be $f(T', C')=f(T, C)=y$. But the output is not $[y]_{\mathcal{E}}$ but $[y+7]_{\mathcal{E}}$, so the method does not work. It does not exist.

Now suppose for contradiction that the operator deliberately builds a new program $C'=f(T, C)$ that returns outputs $[y]_{\mathcal{E}}$ where $y$ is known to and decided by the operator. Then the constants $[k]_{\mathcal{E}}$ of $C'$ are found in $C$ because the operator's technique $f$ has no way of arithmetically combining them (the condition (3) means they cannot be combined arithmetically in the processor nor taken from the trace, and the operator does not have the encryption key). But the argument just given above says the operator cannot read the outputs $[y]_{\mathcal{E}}$ of $C'$, and that is a contradiction. □

It turns out that knowing the code and/or trace does not enable a single bit of the runtime data under the encryption to be guessed with any degree of statistical accuracy. The intuition that explains that is that the number '7' that the proof shows the data might logically differ from the 'nominal' by at any point is arbitrarily chosen. Any possible offset, like '7', at any point in the program, for any particular register or memory location, is as probable as another.

But that does not take into account that human beings only write certain programs, so one can bet, for example, on finding an encrypted 1 as a constant in nearly any program. Practical obfuscation requires a compiler strategy that makes use of the possibilities to really spread the differences from nominal around randomly and uniformly, else all is bluff.

## 3 Obfuscating compilation to FxA

To make compilation to FxA really use the possibilities for obfuscation, not just set the instruction constants to (encrypted) zero every time, the compiler may set a different *offset* $\Delta_{x_l}$ for the value $[x_l]_{\mathcal{E}}$ in each register and memory location $l$, at different points in the program. The offset represents by how much the decrypted data value in the location is to differ from the nominal value (without obfuscation) at runtime. Each instruction that writes at location $l$ offers an opportunity for the compiler to reset the offset $\Delta_{x_l}$ used.

For example, to compile a boolean-valued computational conjunction expression $A \,\&\&\, B$ in the program source code, the possible additive offset is 0 or 1 mod 2. An offset of 0 means the result is returned as is, 'telling the truth'. An offset of 1 means the result is inverted: 'lying'. The compiler chooses between:

(a) whether $A$ is compiled telling the truth or lying
(b) whether $B$ is compiled telling the truth or lying

(c) whether it will lie or tell the truth for $C = A\&\&B$.

The (a) corresponds to whether $\Delta_A$=0 (truth) or $\Delta_A$=1 (liar) is added mod 2 to the result for $A$. Similarly (b) corresponds to whether $\Delta_B$=0 (truth) or $\Delta_B$=1 (liar) is added in to the result for $B$. Let $a$ be true when (a) is set to tell truth ($\Delta_A$=0), and false when (a) is set to lie ($\Delta_A$=1). Similarly for $b$ with respect to (b), and $c$ with respect to (c). Then what should be 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

$$\begin{array}{ll} \text{if } a\,b\,c \text{ then } A\&\&B & \text{if } a\,b\,\overline{c} \text{ then } \overline{A\&\&B} \\ \text{if } \overline{a}\,b\,c \text{ then } \overline{A}\&\&B & \text{if } \overline{a}\,b\,\overline{c} \text{ then } \overline{\overline{A}\&\&B} \\ \text{if } a\,\overline{b}\,c \text{ then } A\&\&\overline{B} & \text{if } a\,\overline{b}\,\overline{c} \text{ then } \overline{A\&\&\overline{B}} \\ \text{if } \overline{a}\,\overline{b}\,c \text{ then } \overline{A}\&\&\overline{B} & \text{if } \overline{a}\,\overline{b}\,\overline{c} \text{ then } \overline{\overline{A}\&\&\overline{B}} \end{array}$$

where the overline means boolean negation. The compiler knows $a$ and $b$ and chooses $c$ with 50/50 probability, deciding which of $A\&\&B$, $\overline{A}\&\&B$, etc., it will generate machine code for. All the generated codes look the same, modulo the encrypted constants, unreadable by the operator. If $[A]$ is the compiled code for $A$ and $[B]$ is the compiled code for $B$, then the compiler produces in every case a machine code sequence $[C]$:

$$[A];\ i_a;\ \text{bnf } l;\ [B];\ i_b;\ l : i_c$$

where if $a$ is true ('truth teller') then for $i_a$ a machine code sequence is generated by the compiler that maintains the flag set by $A$ that the **bnf** instruction tests. It would suffice to emit nothing, but it is required that the sequence look the same for all possible cases, and 'nothing' would be a give-away that that portion of the compilation has been carried out honestly. If $a$ is false ('liar') then $i_a$ is a machine code sequence of the same length that flips the flag set by $A$, the compilation of $A$ having been such that it deliberately gives the 'wrong' result. Whatever the details of the truth/liar compilation decisions in the internals of $A$ and $B$, the code produced has the same length, so the length of the jump in the branch instruction (here represented via the assembler label $l$, not the numerical value later inserted) is always the same.

Apart from the possibly flag-flipping inclusions $i_a$, $i_b$, $i_c$, 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 $A$ (possibly flipped by $i_a$) fails.

The machine code for the $i_a$, $i_b$, $i_c$ is in each case one of the two possibilities:

$$\begin{array}{ll} \text{bf } l_1;\ \text{snf};\ \text{b } l_2;\ l_1 : \text{sf};\ l_2 : & \#\ \text{keep flag} \\ \text{bf } l_1;\ \text{sf};\ \text{b } l_2;\ l_1 : \text{snf};\ l_2 : & \#\ \text{flip flag} \end{array}$$

where **sf** and **snf** are equal-length codes that look alike apart from encrypted constants. They respectively set and clear the flag that is tested by conditional branch instructions. They are:

$$\begin{array}{ll} \text{sflt } r0\ r0\ [1]_{\mathcal{E}} & \#\ \text{sf (set flag)} \\ \text{sflt } r0\ r0\ [0]_{\mathcal{E}} & \#\ \text{snf (clear flag)} \end{array}$$

where $r0$ can be any register. The '1' in **sf** can be any positive value, and the 0 in **snf** may be any non-positive value. Encryption in any case produces different values for $[0]_{\mathcal{E}}$ and $[1]_{\mathcal{E}}$ at every invocation, because of random padding/blinding.

The way to compile the computational disjunction $A \,||\, B$ is similarly

$$[A];\ i_a;\ \text{bf } l;\ [B];\ i_b;\ l : i_c$$

replacing the **bnf** from the conjunction compilation with a **bf** instruction. To compile source code constant expressions 'true' and 'false' the compiler arbitrarily emits the **sf** or the **snf** code, remembering its truth-teller/liar choice for the compilation of the expression of which those form a part.

This compiler emits code that generates runtime values that are arbitrarily different in each register and memory location for each unique compilation, yet the compiled codes (and the traces) look exactly the same, up to the encrypted constants (and runtime data), between one compilation and the next.

## 4 Compilation in general

The compiler works with a database $D : \mathrm{DB} = \mathrm{Loc} \to \mathrm{Int}$ containing (32-bit) integer offsets indexed per register or memory location. As the compiler works through the source code, the offset represents by how much the data underneath the encryption is to vary from nominal at runtime at that point in the program.

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

$$\mathbb{C}^L[\_ : \_] : \mathrm{DB} \times \mathrm{source\_code} \to \mathrm{DB} \times \mathrm{machine\_code}$$

To simplify the presentation here, details of the management of database $L$ are omitted. It is used to look up the location to which a source code variable corresponds. A pair in the cross product is written $D : s$ here.

*Sequence:* The compiler works left-to-right through a source code sequence:

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

The database $D_1$ that results from compiling the left sequent $s_1$ in the source code, emitting machine code $m_1$, is passed in to the subsequent compilation of the right sequent $s_2$, emitting machine code $m_2$ that follows on directly from $m_1$ in the object code file and its memory image when loaded.

*Assignment:* An opportunity for new obfuscation arises at an assignment to a source code variable $x$. An offset $\Delta_x = D_1 Lx$ for the data in the target register or memory location $Lx$ is generated randomly, replacing the old offset $D_0 Lx$ that previously held for the data at that location. The compiler emits code $m_1$ for the expression $e$ which puts the result in a designated temporary location **t0** with offset $\Delta_e = D_1\mathbf{t0}$. It is transferred from there to the location $Lx$ by a following add instruction (the **zer** location dummy in the add instruction means that field does not contribute):

$$\begin{aligned}\mathbb{C}^L[D_0 : x{=}e] &= D_1 :\ m_1; \text{add } Lx\ \mathbf{t0}\ \mathbf{zer}\ [i]_{\mathcal{E}} \\ &\text{where } i = \Delta_x - \Delta_e \\ D_1 : m_1 &= \mathbb{C}^L_{\mathbf{t0}}[D_0 : e]\end{aligned}$$

The **t0** subscript for the expression compiler tells it to aim at location **t0** for the result of expression $e$. That is one of the registers reserved for temporary values.

*Return:* The compiler at a 'return $e$' from function $f$ selects a final offset $\Delta_{f_{\text{ret}}}$ (functions $f$ are subtyped by offsets $\Delta_{f_{\text{par0}}}$, $\Delta_{f_{\text{par1}}}$, etc. in their formal parameters and $\Delta_{f_{\text{ret}}}$ in their return value) and emits an add instruction with target the standard function return value register **v0** prior to the conventional function trailer (ending with a jump back to the address in the return address register **ra**). The add instruction adjusts to the offset $\Delta_{f_{\text{ret}}}$ from the offset $\Delta_e = D_1\mathbf{t0}$ with which the result from $e$ in **t0** is computed by the code $m_1$ compiled for $e$:

$$\begin{aligned}\mathbb{C}^L[D_0 : \text{return } e] &= D_1 :\ m_1; \text{add } \mathbf{v0}\ \mathbf{t0}\ \mathbf{zer}\ [i]_{\mathcal{E}} \\ &\qquad\qquad \ldots \quad \# \text{ restore stack} \\ &\qquad\qquad \text{jr } \mathbf{ra} \ \# \text{ jump return} \\ &\text{where } i = \Delta_{f_{\text{ret}}} - \Delta_e \\ D_1 : m_1 &= \mathbb{C}^L_{\mathbf{t0}}[D_0 : e]\end{aligned}$$

The offset accounted for **v0** is updated in $D_1$ to $D_1\mathbf{v0}{=}\Delta_{f_{\text{ret}}}$.

The remaining source code control constructs are treated like return in the way they adjust the final offset to meet constraints. For an if statement, for example, final offsets in each branch are adjusted to match at the join.

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

*Proof.* Suppose that just before the FxA instruction $I$ in the program, for all locations $l$ the value $x{+}\Delta_x$ with $[x{+}\Delta_x]_{\mathcal{E}}$ in $l$ varies randomly across recompilations with respect to a 'standard' value $x$ with probability $p(x{+}\Delta_x{=}X){=}1/2^{32}$, and $I$ writes value $[y]_{\mathcal{E}}$ in one particular location $l$. That $y$ has an additive component $k$ that is generated by the compilation so as to offset $y$ from the nominal functionality $f(x + \Delta_x)$ by an amount $\Delta_y$ that is uniformly distributed across the possible range. 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 $I$ is only generated once by the compiler and $\Delta_y$ is newly introduced for it), so that sum is $p(y{=}Y){=}\sum_{Y'} p(f(x{+}\Delta_x){=}Y')p(\Delta_y{=}Y{-}Y')$. That is $p(y{=}Y){=}\frac{1}{2^{32}}\sum_{Y'} p(f(x{+}\mathrm{d}x){=}Y')$. Since the sum is over all possible $Y'$, the total of the summed probabilities is 1, and $p(y{=}Y){=}1/2^{32}$. The distribution of $x{+}\Delta_x$ in other locations is unchanged. □

Another intuition is that $\Delta_y$ has maximal entropy, so adding it in in an instruction swamps any other information the instruction might expose.

The theorem states that code is obfuscated. Having the machine code in hand does not tell the operator which of the many compilations of the program it might be, and the data under the encryption at runtime can vary arbitrarily and uniformly across recompilations without changing the trace or the code, as far as a malfeasant who does not have the encryption key is concerned.

Could FxA fused arithmetic machine code instructions, which may be costly at runtime, be done without? No, a single classical arithmetic machine code instruction without the addend would introduce a weakness: the operator has access – via a debugger, for example – to running code, so every machine instruction can be observed and experimented with. The action may be repeatedly observed to build up an encrypted arithmetic table. Then the operator might come across, for example, two encrypted additions '$a{+}b{=}c$' and '$c{+}b{=}a$', from which it may be deduced that $2b{=}0 \pmod{2^{32}}$, so $b{=}0$ or $b{=}2^{31}$. Observing instead an FxA addition instruction only permits $2(b{+}k){=}0$ to be deduced, where $k$ is an unknown extra addend, which says nothing about $b$.

The argument of Theorem 2 works in a more general context:

**Theorem 3.** *A correctly compiled program in an instruction set managing encrypted data and satisfying requirement (1) 'obfuscated instructions'* in context in the program*, is such that the probability across different compilations that any particular runtime 32-bit value $x$ for $[x]_{\mathcal{E}}$ is in location $l$ at any given point in the program is uniformly* $1/2^{32}$*.*

*Proof.* The argument of Theorem 2 may be repeated. Obfuscation 'in context in the program' means that an instruction $I$ that nominally writes a value $y$ to location $l$ can by that hypothesis be varied by the compiler to write a value $y + \Delta_y$ instead that is uniformly distributed across recompilations (while retaining correctness of the compilation). □

Resuming the argument, the compiler is required to collaborate in producing the obfuscation by the hypothesis that each instruction in the program is obfuscated *in context* in the program. That means the output of each instruction cannot be predicted or guessed with any statistical accuracy, so it must vary uniformly across recompilations. Since processors are deterministic, that means the instruction must vary comprehensively but invisibly to the observer, by dint of some encrypted hence effectively hidden parameters that control its behaviour

and which the compiler varies. The compiler defined in this section does that by controlling the offset $\Delta_y$ in the result $y$ of the instruction. By the well-known 'Shannon's inequality' of information theory for the case when $\Delta_y$ has maximal entropy, that implies $y+\Delta_y$ has maximal entropy too, which is the desired result, as maximal entropy means uniformly probable distribution across the range.

## 5 Example

The Ackermann function [15] written in C 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));
};
```

It is a classic function for studying computationally complex behaviour. Every increment in its first argument produces an exponential increase in complexity with respect to the second argument, $A(m,n) = 2^{\cdots 2^{n+3}} - 3$ where the ellipsis covers $m-2$ exponentiations. The first exponential case is $A(3,n) = 2^{n+3} - 3$.

Compilation[2] produces the machine code in Table 2. Places where a 0 constant is nominal (instructions 5, 8, 14, 26, 29, 35) do not all contain a 0 (the literal 0s and 1s in the keep and flip flag macros are for recognisability in testing). Similarly for places where a 1 constant is nominal.

At instruction 38, for example, the offset from the nominal value 1 in register **t1** is $-587705998-1 = -587705999$, $-587705998$ being written, but two instructions later at instruction 40, when 1 is written into the same register again, the offset from nominal is $1111468055 - 1 = 1111468054$, 1111468055 being written.

For testing, the offset of the return value from functions and the offsets for function parameters have been set at 0, but they may be arbitrary random values known only to the compiler. The entry and exit offsets for a complete executable program, on the other hand, must be known to the remote user who compiled the code and commissioned its execution. The trace of a run for $A(2,1) = 5$ is shown in Table 3, with the result in register **v0** after 1104 steps. The $A(3,1)$ computation reaches result 13 in 8288 steps.

Although both stack pointer (register **sp**) and the base addresses held in registers for load memory and store memory instructions (not shown in Table 2) are also obfuscated, so might possibly access invalid RAM addresses, that is not a problem in practice because processors that work with encrypted data are engineered to cope with exactly that. Data addresses are data too, and, being encrypted on those platforms, are distributed over the whole of the possible range, which might be only partially backed by physical RAM. RAM, on the other hand, like any other external local hardware device, must remain ignorant

[2] Haskell source code for the compiler and a virtual machine, including this example, may be downloaded from http://nbd.it.uc3m.es/~ptb/obfusc_comp-0_7.hs.

**Table 2.** Compiled FxA encrypted machine code for the Ackermann function.

```
A:  ...                     # create frame                              # return  A(m-1,1)
                            # if (m == 0)       37  add  t0  a0  zer [-457757118]ε    # m
4   add  t0  a0  zer [-1704185953]ε   # m       38  add  t1  zer zer [-587705998]ε    # 1
5   add  t1  zer zer [2104023132]ε    # 0       39  sub  t0  t0  t1  [-2141902894]ε   # -
6   sfeq t0  t1      [486758211]ε     # ==      40  add  t1  zer zer [1111468055]ε    # 1
7   bf 2                    # keep flag         ...                       # save regs
8   sflt zer zer     [0]ε   # keep flag         ...                       # fill args
9   b 1                     # keep flag         50  jal 0                 # call  A(m-1,1)
10  sflt zer zer     [1]ε   # keep flag         51  add  t0  v0  zer [-1607308215]ε
11  bf 2                    # flip flag         ...                       # restore regs
12  sflt zer zer     [1]ε   # flip flag         57  add  v0  t0  zer [1607308215]ε
13  b 1                     # flip flag         ...                       # destroy frame
14  sflt zer zer     [0]ε   # flip flag         ...                       # return sequence
15  bnf 9                   # then              62  b 0                   # else skip
                            # return n + 1                                # return A(m-1,A(m,n-1))
16  add  t0  a1  zer [1526256091]ε    # n       63  add  t0  a0  zer [1195221673]ε    # m
17  add  t1  zer zer [1280102991]ε    # 1       64  add  t1  zer zer [-868884270]ε    # 1
18  add  t0  t0  t1  [-620124265]ε    # +       65  sub  t0  t0  t1  [-1733760489]ε   # -
19  add  v0  t0  zer [2108732479]ε              66  add  t1  a0  zer [-1996082249]ε   # m
 ...                        # frame destroy     67  add  t2  a1  zer [-1268351424]ε   # n
 ...                        # return sequence   68  add  t3  zer zer [1148618604]ε    # 1
24  b 0                     # else skip         69  sub  t2  t2  t3  [752318999]ε     # -
                            # if (n == 0)       ...                       # save regs
25  add  t0  a1  zer [-989123886]ε    # n       ...                       # fill args
26  add  t1  zer zer [-580299623]ε    # 0       79  jal 0                 # call A(m,n-1)
27  sfeq t0  t1      [-408824263]ε    # ==      80  add  t1  v0  zer [-191727838]ε
28  bf 2                    # keep flag         ...                       # restore regs
29  sflt zer zer     [0]ε   # keep flag         ...                       # save regs
30  b 1                     # keep flag         ...                       # fill args
31  sflt zer zer     [1]ε   # keep flag         95  jal 0                 # call A(m-1,..)
32  bf 2                    # flip flag         96  add  t0  v0  zer [1566613208]ε
33  sflt zer zer     [1]ε   # flip flag         ...                       # restore regs
34  b 1                     # flip flag         102 add  v0  t0  zer [-1566613208]ε
35  sflt zer zer     [0]ε   # flip flag         ...                       # destroy frame
36  bnf 26                  # then              ...                       # return sequence
```

of the encryption in use because it might be subverted [12]. In consequence, some circulating addresses might not be backed by RAM.

But, on those platforms, data addresses are linearly remapped to backed regions on a first-come, first-served basis within the processor. The encrypted addresses serve only as labels. The physical RAM location to which they correspond is defined internally by the 'translation lookaside buffer' (TLB) hardware in the processor, itself backed in RAM, and cached. The effect is to keep memory references pointing, via the TLB mapping, into physically backed space.

Nevertheless, the nondeterministic nature of good encryption means that repeating the same address under the encryption does not necessarily invoke the same encrypted address. Correction is best made at software level, compiling so encrypted addresses are saved and copied, not recalculated, when reused [4].

It might be thought that the $A(2,1)$ result, say, could be recognized from the form of the code and the control flow in the trace, but there is nothing there to distinguish from the Ackermann function plus 7, for example (Theorem 1).

**Table 3.** Runtime trace (abridged) for the Ackermann function on (2,1), result 5.

| PC | instruction | update | flag |
|---|---|---|---|
| ... | | | |
| 4 | add t0 a0 zer $[-1704185953]_{\mathcal{E}}$ | t0 = $[-1704185951]_{\mathcal{E}}$ | 0 |
| 5 | add t1 zer zer $[2104023132]_{\mathcal{E}}$ | t1 = $[2104023132]_{\mathcal{E}}$ | 0 |
| 6 | sfeq t0 t1 $[486758211]_{\mathcal{E}}$ | | 0 |
| 7 | bf 2 | | 0 |
| 8 | sflt zer zer $[0]_{\mathcal{E}}$ | | 0 |
| 9 | b 1 | | 0 |
| 11 | bf 2 | | 0 |
| 12 | sflt zer zer $[0]_{\mathcal{E}}$ | | 0 |
| 13 | b 1 | | 0 |
| 15 | bnf 9 | | 0 |
| 25 | add t0 a1 zer $[-989123886]_{\mathcal{E}}$ | t0 = $[-989123885]_{\mathcal{E}}$ | 0 |
| 26 | add t1 zer zer $[-580299623]_{\mathcal{E}}$ | t1 = $[-580299623]_{\mathcal{E}}$ | 0 |
| 27 | sfeq t0 t1 $[-408824263]_{\mathcal{E}}$ | | 0 |
| 28 | bf 2 | | 0 |
| ... | | | |
| 102 | add v0 t0 zer $[-1566613208]_{\mathcal{E}}$ | v0 = $[5]_{\mathcal{E}}$ | 1 |
| ... | | | |
| 106 | jr ra | | 1 |
| STOP | | | |

## Conclusion

This paper has considered the privacy of remote encrypted computation with respect to the operator/administrator of the cloud-based server as adversary. A minimally encrypted machine code instruction set for computation on the server has been defined that fuses arithmetic operations with the addition of one or more constants. It has been shown that the instruction set architecture allows code and program traces and (encrypted) data circulating in the processor to be accessible to the operator in the conventional way, while being formally private for the remote user. An 'obfuscating compiler' has been defined that provides uniformly distributed runtime variations across different recompilations at every point in the program trace. It supports the formal assurances of privacy by eliminating the possibility of attacks based on the likely use by a human author of small numbers in program or data. It does not contradict Barak et al.'s famous result that program obfuscation is impossible, rather constructs it among a reduced class of machine code programs: those that can be compiled for the special target architecture from the same high-level source by this compiler.

## References


1. Barak, B.: Hopes, fears, and software obfuscation. Commun. ACM 59(3), 88–96 (Feb 2016)
2. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) Proc. 21st Annu. Int. Cryptol. Conf. (CRYPTO'01). pp. 1–18. Adv. Cryptol., Springer (2001)
3. Breuer, P.T., Bowen, J.P.: A fully homomorphic crypto-processor design: Correctness of a secret computer. In: Jürjens, J., Livshits, B., Scandariato, R. (eds.) Proc. 5th Int. Symp. Eng. Sec. Softw. Syst. (ESSoS'13). pp. 123–138. No. 7781 in LNCS, Springer, Berlin/Heidelberg (Feb 2013)

4. Breuer, P.T., Bowen, J.P.: Certifying machine code safe from hardware aliasing: RISC is not necessarily risky. In: Counsell, S., Núñez, M. (eds.) Softw. Eng. Formal Methods. pp. 371–388. No. 8368 in LNCS, Springer, Heidelberg (2014), proc. SEFM Colloc. Workshops (OpenCert'13)
5. Breuer, P.T., Bowen, J.P.: A fully encrypted microprocessor: The secret computer is nearly here. Procedia Comp. Sci. 83, 1282–1287 (Apr 2016)
6. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: A practical encrypted microprocessor. In: Callegari, C., van Sinderen, M., Sarigiannidis, P., Samarati, P., Cabello, E., Lorenz, P., Obaidat, M.S. (eds.) Proc. 13th Int. Conf. Sec. Cryptog. (SECRYPT'16). vol. 4, pp. 239–250. SCITEPRESS, Portugal (2016)
7. Conway, J.H.: Fractran: A simple universal programming language for arithmetic. In: Open Problems in Communication and Computation, pp. 4–26. Springer (1987)
8. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Springer, Berlin/Heidelberg (2002)
9. van Dijk, M., Juels, A.: On the impossibility of cryptography alone for privacy-preserving cloud computing. HotSec 10, 1–8 (2010)
10. Fletcher, C.W., van Dijk, M., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: Proc. 7th ACM Scalable Trusted Comput. Workshop (STC'12). pp. 3–8. ACM, New York (2012)
11. Hada, S.: Zero-knowledge and code obfuscation. In: Okamoto, T. (ed.) Proc. 6th Int. Conf. Theor. & Applicat. Cryptol. and Inform. Sec. (ASIACRYPT'00), pp. 443–457. No. 1976 in LNCS, Springer, Berlin/Heidelberg (2000)
12. Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., Felten, E.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM 52(5), 91–98 (2009)
13. Ostrovsky, R., Goldreich, O.: Comprehensive software protection system (Jun 16 1992), US Pat. 5,123,045
14. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Proc. EUROCRYPT'99. pp. 223–238. Adv. Cryptol., Springer (1999)
15. Sundblad, Y.: The Ackermann function. a theoretical, computational, and formula manipulative study. BIT Num. Math. 11(1), 107–119 (Mar 1971)
16. Tsoutsos, N., Maniatakos, M.: Investigating the application of one instruction set computing for encrypted data computation. In: Gierlichs, B., Guilley, S., Mukhopadhyay, D. (eds.) Proc. Sec., Priv. Appl. Cryptog. Eng. (SPACE'13), pp. 21–37. Springer, Berlin/Heidelberg (2013)
17. Tsoutsos, N., Maniatakos, M.: The HEROIC framework: Encrypted computation without shared keys. IEEE Trans. CAD Integ. Circ. Syst. 34(6), 875–888 (2015)
18. Wang, Z., Lee, R.B.: Covert and side channels due to processor architecture. In: Proc. 2nd Annu. Comp. Sec. Applic. Conf. (ACSAC'06). pp. 473–482. IEEE (2006)
19. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: Proc. ACM Conf. Comp. Commun. Sec. (CCS'12). pp. 305–316. ACM, New York, NY (2012)