On Obfuscating Compilation for Encrypted Computing

Peter T. Breuer[1], Jonathan P. Bowen[2], Esther Palomar[3] and Zhiming Liu[*,4]
[1]*Hecusys LLC, Atlanta, GA, U.S.A.*
[2]*London South Bank University, London, U.K.*
[3]*Birmingham City University, Birmingham, U.K.*
[4]*RISE, Southwest University, Chongqing, China*

Keywords: Obfuscation, Compilation, Privacy, Encrypted Computing.

Abstract: This paper sets out conditions for privacy and security of data against the privileged operator on processors that 'work encrypted'. A compliant machine code architecture plus an 'obfuscating' compiler turns out to be both necessary and sufficient to achieve that, the combination mathematically assuring the privacy of user data in arbitrary computations in an encrypted computing context.

1 INTRODUCTION

A well-known argument (van Dijk and Juels, 2010) equates privacy of user data on a computing platform with *cryptographic obfuscation* (Hada, 2000) against the *privileged operator as adversary*. Inputs, outputs, and any intermediate data that may be accessible are maintained by the processor in the user's personal encryption, or the data would intrinsically be readable by any observer. Then privacy equates formally with obfuscation (the argument will be set out at the start of Section 2). Several prototype processors already support that encrypted mode of working (they will be described in Section 3), allowing operators and operating system alike to see and manipulate user data while keeping it in encrypted form. On such platforms, the operators can single-step the machine, examine data and program instructions in registers and memory, and change anything and everything to which they have access, copying and repeating as required, but the unencrypted form of data is unavailable to them.

In that context, the fundamental question is whether there exists a bona fide computational process that the operator can leverage to produce encryptions of some known numerical values with a degree of certainty. That would enable a 'known plaintext attack' (KPA) against the encryption. A KPA may eventually break the encryption and make accessible the user's data for reading and/or subversion. On the face of it, there is at least one such 'computational process' that does that, because the operator ought to be able to issue an instruction that causes the processor to subtract an encrypted number from itself, yielding an encrypted zero with absolute certainty.

This paper examines the conditions for that to (not) be possible. 'Non-functional' avenues of attack via statistics of cache hits, power consumption, etc. are preventable by engineering means and are not considered here, but there would be in principle nothing to be done about using a processor's computing function as a vulnerability, if such an approach could succeed. It is argued here that a compliant machine code architecture plus an 'obfuscating' compiler are necessary and sufficient to prevent that, and our contribution here is in setting out abstractly how hardware, instruction set and compiler must play together to mathematically assure the privacy and security of data in encrypted computing.

The layout of the paper is as follows. Section 2 analyses the conditions for privacy and Section 3 discusses extant prototype processors that support encrypted running and satisfy the conditions of Section 2 to varying degrees. Section 4 discusses instruction sets satisfying the conditions, and Sections 5 and 6 introduce obfuscating compilers. The latter sections contain the major mathematical results.

2 PRIVACY CONDITIONS

Van Dijk and Juels' 'well-known argument' referred

* Correspondence: Z. Liu, RISE, Southwest University, Tiansheng Rd, Beibei, Chongqing, China

Breuer, P., Bowen, J., Palomar, E. and Liu, Z.
On Obfuscating Compilation for Encrypted Computing.
DOI: 10.5220/0006394002470254
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, pages 247-254
ISBN: 978-989-758-259-2
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

to at the start of Section 1 asks what happens when the encrypted input data for a user's problem and the encrypted code for treating it are both delivered for execution to a virtual machine (VM) running on a platform for encrypted computing. The VM should decrypt the data and the code, with probable hardware assistance via instructions that physically do encryption and decryption as required, and run the code in a private area of the processor with the data as input, probably with interrupts disabled to prevent tampering. The results should be encrypted for return. The question to put is: is this situation more helpful to the hypothetical adversary than running the code on the data on a real machine locked inside a metal safe, with no access for operator and users alike, other than to the (encrypted) inputs and outputs? If the answer is 'no', then (a) that is as good as privacy for computation can ever get, and (b) that is precisely Hada's definition for data to have been effectively cryptographically obfuscated on the processor.[1] That establishes the equivalence of obfuscation with the goal: if the data cannot be attacked any more effectively than on a black box implementation of the program, then the privacy that has been achieved cannot be bettered.[2] So effective obfuscation of data is what to aim for.

Reasoning about the problem begins by noting that there are processors that already do 'obfuscation by design'. An example is the Ascend co-processor (Fletcher et al., 2012), which executes code in Fort Knox-style physical isolation, with no operator access to it while 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 (Wang and Lee, 2006; Zhang et al., 2013), obey statistics that are configured beforehand and have no correlation with the running program, apart from the run time.[3] For good measure, the executable is also input in encrypted form. Since the operator's access to the running program is restricted by a physical black box, the program and data are obfuscated, by the definition given. But is effective obfuscation still possible with a less extreme solution, one that permits debugging, interrupt routines and system calls, for example? Here we require:

Operators and users have conventional access.

That means access to registers and memory, and the user's code may be single stepped, repeated, retried, altered, etc. by the operator.

For obfuscation to go through in those circumstances constrains the processor hardware as follows:

(1) Each machine code instruction is a black box.

That is, individual instructions are treated as Ascend treats a whole program at a time. Only the inputs and outputs from each instruction are visible, in order that they may effectively be a 'black box'. Instructions are executed in a processor in many stages along a pipeline, and those intermediate stages must not be visible. That is functionally so in any standard processor, where interrupts are only enabled at the point where an instruction finishes, but there must also be no clue such as cache hit statistics or power consumption data that may reveal the action of the instruction.

It is usual practice in mathematics to phrase results in relative form. It is assumed here:

The encryption used is secure in its own right.

That is, the reasoning in this paper will suppose encrypted values are unreadable by an adversary. Logically they also cannot be created to order (the writer, knowing them, could 'read' them too). A spy's game is to guess their meaning from observing or interfering with a computation; encryption is not on the line, computation is. If the spy sees $x+x$ and also $x*x$ with identical result, the spy will deduce that x is encrypted 0, 1, or 2, and the danger is of computation leaking information on encrypted data, or subverting it.

The abstraction is realistic for one-to-many encryptions of block size such that the number of instructions that may execute is small relative to the expected interval between cipherspace collisions.

The inputs to and outputs from each instruction must also be in encrypted form, as in Ascend, or they would be readable by the operator:

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

Otherwise, the operator, able to single-step the machine and access registers, can read every action.

This does not mean that inputs and outputs must be in encrypted form all the time: it suffices for the hardware to encrypt when an instruction in privileged mode views a processor register with user data in, for example, or a user mode instruction writes to memory.

Beyond the above, software toolchain support is also required, otherwise the code itself might obviate

[1] This definition of cryptographic obfuscation follows Hada but restricts what is of interest to data, not code.

[2] An argument (Barak et al., 2001) that cryptographic obfuscation as defined by Hada is impossible in the general case does not apply because the inputs and outputs are encrypted here, and hardware assistance is permitted.

[3] 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* (Ostrovsky and Goldreich, 1992), which remaps the logical to physical address relation dynamically, maintaining aliases, so access patterns are statistically impossible to spot. It also masks programmed accesses among independently generated random accesses.

all protections. A human author will write code involving small numbers that can be guessed in a dictionary attack. A kind of *obfuscating compiler* is needed that ensures that the numbers used (encrypted) in code and also the numbers circulating (encrypted) at runtime have no predictable bias that can be leveraged into an attack on the encryption.

The compiler at each recompilation of the same program must produce an executable that in the code and at any point in the runtime *trace* (the sequence of instructions and register/memory states) has *arbitrarily and uniformly distributed differences in the data values* under the encryption with respect to any other compilation. How is that compatible with producing the same answer from the same inputs, no matter how the source code is compiled? It is not, but so long as the compiler agrees with or simply makes known to the owner of the code an offset A on the input and an offset B on the output of the intended functionality $f(x+A)+B$ under the encryption, then that is permissible. The owner of the code must incorporate the extra offset A before encryption of the data for execution, and remove the extra offset B after decryption of the result. So long as A and B are independently randomly generated and uniformly distributed, it follows that the (decrypted) inputs $x+A$ and outputs $y+B$ are too, no matter what biases select x and y, as required.[4]

Considering a single instruction with functionality $y \leftarrow f(x)$ under the encryption, it must be possible for the compiler to vary the functionality to $y \leftarrow f(x+A)+B$ without an observer telling which A and B have been selected, so their distributions remain unbiased with respect to the observer. That is, each instruction must be *malleable* in the following sense:

(3) Each instruction supports arbitrary additions A, B to inputs and outputs via adjustments of (encrypted) parameters in the instruction.

There is just one permissible exception: an instruction that copies a value from one location to another may do so faithfully. An observer will still not be able to rely on any statistical bias in any particular runtime data value, but will know that whatever it is, it is the same as in the datum it has been copied from, if any.

We also wish to consider the situation without the complication of imagining that the operator, acting as adversary, can experiment by taking an encrypted data value from a user program trace and substituting it into instructions of their own devising in order to test it or further modify it using the processor. Nor should the hardware allow the encrypted constants that appear in some instructions to work correctly when used as inputs for arithmetic. That makes it impossible for the spy in the computer room to pass the encrypted constants seen in programs through the processor arithmetic, patching the results back into new code snippets. So the hardware should ensure:

(4) There are *no collisions* between (i) encrypted constants that appear in instructions and (ii) runtime encrypted data values in registers or memory.

That has to be actively enforced in the processor. It is not a logical consequence of assuming secure encryption. It may be achieved in an implementing processor by different padding or blinding factors for the two domains (i-ii), checked in the processor pipeline.

The rest of this paper will use conditions (1-4) to construct a means for obfuscation of user data.

3 SUPPORTING PLATFORMS

As remarked in Section 1, there are prototype platforms that satisfy at least the conditions (1-2). Apart from Ascend, which satisfies them trivially, there is HEROIC (Tsoutsos and Maniatakos, 2015), a prototype 16-bit machine running with a deterministic Paillier (2048 bit) encryption (Paillier, 1999). Its core executes an encrypted addition in 4000 cycles on its fundamentally 200MHz hardware, roughly equivalent to a 25KHz Pentium's speed. Instructions work on encrypted data, producing encrypted data.

The KPU design (Breuer and Bowen, 2014; Breuer and Bowen, 2016; Breuer et al., 2016) generalises HEROIC's approach, achieving encrypted running by modifying the arithmetic logic unit (ALU) for encrypted working in a roughly conventional 32/64/128-bit RISC (Patterson, 1985) processor layout. The modification to the arithmetic causes data to circulate and be processed in encrypted form. A KPU design may embed any encryption the block size of which matches the word size and the hardware for which fits into reasonably many stages of a processor pipeline. Running the US Advanced Encryption Standard (AES) 128 bit encryption (Daemen and Rijmen, 2002) in 10 pipeline stages and using a 1GHz clock it runs at the speed of a 300-500MHz Pentium, broadly comparable with current PCs.[5] Those tests are with

[4] An information theory argument establishes that $x+A$ (mod 2^{32}, say) is randomly and uniformly distributed when A is, no matter what distribution x follows provided it is independent of A. The argument is that A has maximal entropy and $x+A$ cannot have less entropy (Shannon), so must have maximal entropy too, which means it takes randomly and uniformly distributed values across the whole range.

[5] Dhrystones v2.1 rating of 104-140MIPS. A 1GHz 686 family Pentium M is rated at 420MIPS according to the table at www.roylongbottom.org.uk/dhrystone results.htm. A 200MHz 586 family classic Pentium is rated at 48.1MIPS.

contemporary 13.5ns latency memory and 3ns latency cache. The slowdown over unencrypted running is 10-50%. Ascend, running with similar technology, instruction set, and AES-128, slows by 12-13.5×.

The HEROIC design also satisfies condition (3). The 'OI' in HEROIC stands for 'one instruction', meaning that the machine code architecture has only one kind of instruction, a combined 'add, compare, branch' (that is classically computationally complete together with recursion and at least one nonzero constant – the instruction comprises the mathematician J.H. Conway's *Fractran* programming language (Conway, 1987), often used for theoretical studies in computability and complexity). Its instructions may be considered compounds of *addition of a constant* $y \leftarrow x+k$ and branches that *compare with a constant* $x<K$. Addition of a constant satisfies condition (3) because an instruction that has the effect of $y \leftarrow (x+A)+k+B$, encrypted, can be obtained by running $y \leftarrow x+k'$, encrypted, where k' is $k+A+B \bmod 2^{32}$. Compare with a constant also satisfies the condition, because an instruction that compares $x+A < K$ can be obtained by running an instruction that compares $x < K'$ where K' is $K-A \bmod 2^{32}$ (to agree, the reader needs to know that the conventional 2s complement comparison $u < v$ is invariant under translations).

The KPU runs the standard OpenRISC v1.1 instruction set (see https://opencores.org/or1k/Architecture_Specification) that does not satisfy (3) but it may be modified to do so.

Neither HEROIC nor the KPU support condition (4) at present (that encrypted data should not work as encrypted program constants and vice versa), but it can be arranged in the case of the KPU since the processor has internal access to the decrypted form of data, including the padding under the encryption.

4 COMPLIANT INSTRUCTIONS

Instruction sets are conventionally made up of instructions that (a) perform a relatively simple *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 *comparison operation* such as 'less than' between values in registers or memory, branching to a different point in the program if it is satisfied, plus (c) *unconditional control* instructions such as jump to and return from a subroutine that always alter which instruction is executed next. In the following, we will omit ubiquitous 'under the encryption' qualifiers in the instruction semantics.

Many standard instructions of type (a) such as addition $z \leftarrow x+y$ of two runtime values do not satisfy condition (3). There are no configurable parameters for the compiler to modify its semantics to $z \leftarrow (x+A_1)+(y+A_2)+B$, as condition (3) stipulates. At least one inline constant is required. For addition, a nonstandard instruction with semantics $z \leftarrow x+y+k$ satisfies condition (3), with two operands filled at runtime and one provided by the compiler. The semantics $z \leftarrow (x+A_1)+(y+A_2)+k+B$ may be obtained by $z \leftarrow x+y+k'$ adjusting $k' = k+A_1+A_2+B$.

For multiplication, $z \leftarrow x*y$ does not satisfy condition (3) and neither does $y \leftarrow x*k$, the two standard instructions. However, AMD and Intel in 2011-2013 introduced so-called *fused* instructions that combine *two* arithmetic operations. While addition takes one cycle to complete in a processor, multiplication takes much longer (about ten cycles), and the repeating subunit that forms the long 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 feed two full integer addends into the calculation at no extra cost, and such a 'fused multiply and add' (FMA) instruction was introduced in AMD and Intel's FMA3 and FMA4 instruction sets for reasons of efficiency. A fused multiply and add instruction satisfying (3) is, for example, $x_3 \leftarrow (x_1-k_1)*(x_2-k_2)+k_3$. In general, if instruction semantics is $x_2 \leftarrow f(x_1)$, then condition (3) says that changing x_1 by A and x_2 by B is achieved by modifying constant parameters, so the full instruction semantics must be $x_2 \leftarrow f(x_1-k_1)+k_2$.

A branch instruction (b) compares two operands $x_1<x_2$. Condition (3) requires there be parameters in the instruction for the compiler to generate the semantics of $(x_1-k_1)<(x_2-k_2)$ and $(x_1-k_1) \not< (x_2-k_2)$ from it. That may be done by emitting different instructions with tests of the form $x_1<x_2+k'$ and $x_1 \geq x_2+k'$ with $k' = k_2-k_1$. In conclusion, a complete set of machine code instructions compliant to (3) is very feasible.

Then consider non-probabilistic attacks on a program C running in a processor working encrypted and satisfying (1-4), with instructions satisfying (3). Recall the encryption is assumed secure in its own right:

Theorem 1. *There is no deterministic method by which the privileged operator can read the encrypted data read or written by program C, nor alter C to write an intended encrypted value.*

Proof. (Sketch) Imagine that every runtime datum circulating in the processor under the encryption has magically increased by 7. We will adapt the program C by changing some of its embedded encrypted constants to accept that change while still generating a trace T that looks the same as before, up to encryption. The supposed method sees the new program and

trace as the same as before and reports the old answer, whereas it should report the old answer plus 7. Done.

(Detail) Suppose for contradiction that the operator has a method $f(T,C)=y$ of finding that the output $[y]_{\mathcal{E}}$ of C encrypts y, having observed the trace T, a sequence of states s_i of the register/memory with instruction p_i at address a_i transforming $s_i \stackrel{a_i:p_i}{\to} s_{i+1}$. Define a transform $s \to s'$ by adding 7 to the number in each location r in state s giving the state s' with $s'r = [[s\,r]_{\mathcal{D}}+7]_{\mathcal{E}}$, $[\cdot]_{\mathcal{D}}$, $[\cdot]_{\mathcal{E}}$ representing de/encryption.

A program C' is obtained by (i) replacing every instruction p at address a in C that is of the form $r_0 \leftarrow [([r_1]_{\mathcal{D}}-k_1)\,\Theta([r_2]_{\mathcal{D}}-k_2)+k_0]_{\mathcal{E}}$ with p' of the same form $r_0 \leftarrow [([r_1]_{\mathcal{D}}-k'_1)\,\Theta([r_2]_{\mathcal{D}}-k'_2)+k'_0]_{\mathcal{E}}$ where $k'_i = k_i+7$, $i=0,1,2$; (ii) if the instruction is a branch with test $([r_1]_{\mathcal{D}}-k_1)R([r_2]_{\mathcal{D}}-k_2)$, then it is likewise changed to $([r_1]_{\mathcal{D}}-k'_1)R([r_2]_{\mathcal{D}}-k'_2)$ with $k'_i = k_i+7$, $i=1,2$; (iii) unconditional jump instructions p at address a are left just as they are, with $p'=p$. The replacements (i) are designed so $s' \stackrel{a:p'}{\to} t'$ where $s \stackrel{a:p}{\to} t$. The next instruction address is $a+1$. Branches (ii) with a test $([r_1]_{\mathcal{D}}-k'_1)\,R([r_2]_{\mathcal{D}}-k'_2)$ take the same jump at state s' as $([r_1]_{\mathcal{D}}-k_1)\,R([r_2]_{\mathcal{D}}-k_2)$ does at state s, getting the same result from the test. The unconditional jumps (iii) also take the same jump at state s' as at state s.

Thus if T with $s_i \stackrel{a_i:p_i}{\to} s_{i+1}$ is a trace of C, then T' with $s'_i \stackrel{a_i:p'_i}{\to} s'_{i+1}$ is at least a feasible trace of C', hence is the unique trace of C' in the deterministic processor.

The program C' 'looks the same', $C' \sim C$, differing only by the encrypted numbers $[k]_{\mathcal{E}}$ instead of $[k']_{\mathcal{E}}$ in it. The new program trace T' 'looks the same' too, $T' \sim T$, in the same sense, that is, up to the encrypted numbers in it, because the branches after comparisons go the same way as in T and make the same jump, giving rise to the same sequence of instruction addresses. The difference is states s'_i instead of s_i, and instruction p'_i instead of p_i at address a_i, and the differences between all those are different encrypted numbers.

If the method f were sensitive to differences in encrypted values then it would vary while the value underneath the encryption stayed constant[6], as it cannot read the encryption by hypothesis. So it must be constant with respect to changes in encrypted values, with $C \sim C'$ and $T \sim T'$ implying $f(T,C) = f(T',C')$. So $f(T',C')=f(T,C)=y$. Yet the output of C' is not $[y]_{\mathcal{E}}$ but $[y+7]_{\mathcal{E}}$, so the method does not work.

[6]Formally it should be supposed that any feature of encrypted values that the attacker's probes f may be sensitive to, such as counting the number of 7s, is triggered by some encryption $[y]_{\mathcal{E}}$ of *every* value y, so $f(\ldots,[y]_{\mathcal{E}},\ldots)$ must be constant with respect to that parameter if f really detects y.

For the second half of the result, suppose the operator 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 in C because the operator's technique f has no way of arithmetically combining them (the condition (4) means they cannot be combined arithmetically in-processor nor taken from the trace, and the operator does not have the encryption key). The first half of the result now says the operator cannot read the outputs $[y]_{\mathcal{E}}$ of C', a contradiction. □

Remark 1. *The proof inspected in detail shows that runtime data through instructions satisfying condition (3) (case (i) in the proof) in the trace can be altered independently via the constants in the instructions.*

So intuitively, knowing the instructions and/or trace ought not to enable any bit of the runtime data under the encryption to be guessed with any degree of statistical accuracy because the program might have been modified to have the data offset by any amount from what might naturally have been guessed. But is any possible offset from nominal, at any point in the program, for any particular register or memory location, *equally* as probable as another? That cannot be said without the probabilistic setting of Section 6.

In particular that does not take into account that human beings only write certain programs, so one can bet, for example, on finding an encrypted 1 in nearly any program trace. Leveraging the intuition above into a practical tool for obfuscation requires a compiler strategy, described in the next sections, that varies compiled code so runtime data varies randomly and uniformly from nominal values, else all is bluff.

5 OBFUSCATING COMPILATION

To make compilation to a compliant instruction set as described in Section 4 use the possibilities for obfuscation in order to frustrate a dictionary attack against the runtime data values, the compiler should set an arbitrary *offset* Δ_{x_l} for x_l, where the value $[x_l]_{\mathcal{E}}$ is in the register or memory location l, at different points in the program. This is manipulated by the compiler. The offset represents by how much the decrypted data value x_l in the location is to vary from nominal (without obfuscation) at runtime. Each instruction that writes at location l offers an opportunity for the compiler to reset the offset Δ_{x_l} used there.

For example, in compiling 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 Δ_A=0 (truth) or Δ_A=1 (liar) is added mod 2 to the result for A. Similarly (b) corresponds to whether Δ_B=0 (truth) or Δ_B=1 (liar) is added in to the result for B. Let a be **true** when (a) is to tell truth (Δ_A=0), and **false** when (a) is to lie (Δ_A=1). Similarly for b with respect to (b) and c with respect to (c). What is to be computed at runtime is:

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

where the arrow in $x \leftrightarrow y$ stands for the boolean biconditional operator. That is:

if abc then $A\&\&B$	if $ab\overline{c}$ then $\overline{A\&\&B}$
if $\overline{a}bc$ 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 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 will look alike, modulo the encrypted constants, unreadable by the operator. If $[A]$ is the compiled code for A and $[B]$ is the compiled code for B, producing values 1/0 for **true/false** respectively in register **t0**, then the compiler emits a machine code sequence $[C]$:

$$[A];\ i_a;\ \textbf{beqi t0}\ [0]_{\mathcal{E}}\ l;\ [B];\ i_b;\ l : i_c$$

where if a is **true** ('truth teller') then i_a is the machine code sequence that maintains the value set by A in register **t0** that the **beqi** instruction tests against the zero supplied as an encrypted constant, and jumps to the point l if equal, 'short-circuiting' the calculation. It suffices to emit nothing, but it is required that the sequence look the same for all possible cases, and 'nothing' would be a give-away. If a is **false** ('liar') then i_a is a machine code sequence of the same length that flips the value set by A, the compilation of A being such that it deliberately gives the 'wrong' result.

The final i_c sequence can move to both branches:

$$[A];\ i_a;\ i_c;\ \textbf{beqi t0}\ [\overline{c}]_{\mathcal{E}}\ l;\ [B];\ i_b;\ i_c;\ l :$$

The sequence $i_a;i_c$ either flips the value no times, once, or twice, the last being the same as no times, so does the same as $i_{a\leftrightarrow c}$ and may be replaced by it:

$$[A];\ i_{a\leftrightarrow c};\ \textbf{beqi t0}\ [\overline{c}]_{\mathcal{E}}\ l;\ [B];\ i_{b\leftrightarrow c};\ l :$$

To avoid exposing via $[\overline{c}]_{\mathcal{E}}$ an encryption of 0 or 1, the $i_{a\leftrightarrow c}$ code may produce a result that is offset by a compiler-decided random constant k from nominal, and then the branch tests against $\overline{c}+k$ instead of $\overline{c}$:

$$[A];\ i^k_{a\leftrightarrow c};\ \textbf{beqi t0}\ [\overline{c}+k]_{\mathcal{E}}\ l;\ [B];\ i_{b\leftrightarrow c};\ l :$$

The codes for A and B always have the same length and form, differing only in inline encrypted constants, so that is true of the whole too. There is no possibility of an observer spotting when the value in **t0** is left unchanged and when it is changed by $i^k_{a\leftrightarrow c}$ because the encrypted value is always changed, even when the decrypted value is not.

Let the instruction **xori** x_2 x_1 k k_2 k_1 have semantics $x_2 \leftarrow (x_1 - k_1)\hat{}k + k_2$, computing *bitwise exclusive or* ('xor'). The code for the i_* can be one of:

xori t0 t0 $[0]_{\mathcal{E}}$	$[0]_{\mathcal{E}}$ $[0]_{\mathcal{E}}$	# keep bool value
xori t0 t0 $[-1]_{\mathcal{E}}$	$[0]_{\mathcal{E}}$ $[0]_{\mathcal{E}}$	# flip bool value

since x XOR 1 is the complement of the boolean value x. The code for $i^k_{a\leftrightarrow c}$ is one of:

xori t0 t0 $[0]_{\mathcal{E}}$	$[k]_{\mathcal{E}}$ $[0]_{\mathcal{E}}$	# keep bool value
xori t0 t0 $[-1]_{\mathcal{E}}$	$[k]_{\mathcal{E}}$ $[0]_{\mathcal{E}}$	# flip bool value

The $[0]_{\mathcal{E}}$ and $[-1]_{\mathcal{E}}$ will also be offset by values arbitrarily chosen by the compiler via a further application of the 'shift by k' technique during the compilation of A and B, as set out in the next section.

6 COMPILING STATEMENTS

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

The compiler also maintains a database $L : \text{Var} \rightarrow \text{Loc}$ of the location for each source code variables (type Var) placement in registers or memory. Let DB abbreviate the type of database D, then the compiler has type signature:

$$\mathbb{C}^L[_ : _] : \text{DB} \times \text{source_code} \rightarrow \text{DB} \times \text{machine_code}$$

As syntactic sugar, a pair in the cross product is written $D : s$, or $D : m$, and details of the entirely conventional management of database L are omitted here.

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

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

The database D_1 that results from compiling the left sequent s_1 in the source code, emitting machine code m_1, is passed to the compilation of the right sequent s_2, emitting machine code m_2 following on from m_1.

Assignment: An opportunity for new obfuscation arises at any assignment to a source code variable x. An offset $\Delta_x = D_1Lx$ for the data in the target register or memory location Lx is generated randomly, replacing the old offset D_0Lx 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. Let the machine code instruction '**addi** $r_2\, r_1\, [i]_{\mathcal{E}}$' have semantics $x_2 \leftarrow x_1{+}i$ where the content of register r_2 is $[x_2]_{\mathcal{E}}$ and the content of register r_1 is $[x_1]_{\mathcal{E}}$. Then the emitted code is

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

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. The add instruction in the trailer 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:

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

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

Other source code control constructs are treated like return in the way they adjust the final offset to meet constraints. For an if statement, final offsets in each branch are adjusted to match at the join. A while statement is an if statement that joins back to its own start, so the final offset in the loop must equal the initial offset. Each function definition is compiled separately, the databases being flushed before each.

Remark 2. *The* **addi** *instructions in the cases (b-c) above contain an embedded value i that is contributed to by a freely chosen constant* $k{=}\Delta_x$ *in (b) and* $k{=}\Delta_{f_{\text{ret}}}$ *in (c), which will be referred to generically as k in the proof of the theorem below. It is chosen by the compiler from a distribution designed to make uniform the distribution of values written by the instruction.*

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

Proof. 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 l' other than that written by I and at all other points in the program to vary the value $x_{l'} = x{+}\Delta_x$ with $[x_{l'}]_{\mathcal{E}}$ in l' randomly and uniformly across compilations, taking advantage of the possibilities in the instruction set, as exhibited in the compiler specification. Let I write value $[y]_{\mathcal{E}}$ in location l. By condition (3) (c.f. the remark above) I has a parameter k that may be tweaked to offset y from the nominal result $f(x+\Delta_x)$ with respect to its input $x+\Delta_x$ by an amount Δ_y. The compiler chooses k with a distribution such that Δ_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 Δ_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 Δ_y is newly introduced), 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 $x_{l'} = x{+}\Delta_x$ in other locations l' is unchanged. Done by induction on the machine code graph structure. □

A helpful intuition is that Δ_y has maximal entropy, so adding it in swamps all biases in the distribution of y.

The result provides the probabilistic setting for semantic security. Recall that encryption is assumed secure so collisions may be assumed to be avoided. Consider a hypothetical probabilistic method F that guesses for a particular runtime value 'the top bit is 1, not 0', as applied to a compiled program C and its trace T with probability $p > 0.5$ over many trials of the method. In fact, results 1 and 0 are equally likely across all possible compilations according to Theorem 2, and the probability (see below) F is right is

$$0.5(1{-}p) + 0.5p = 0.5 \qquad (*)$$

That is because the method F cannot tell which of the compilations C it is looking at as all the compiled

codes and their traces T are exactly the same modulo the encrypted values in them. There are no collisions, certainly not between program constants and runtime data, as condition (4) maintains, so each compiled code C and trace T consists of different values never repeated internally or between different pairs C, T. All codes C are the same length and form and all traces T are the same length and form (they all branch the same way at the same points). The method F applied to different C and T has nothing to cause it to give different answers except incidental features of the encrypted values (such as the total number of 7s in the decimal representations, perhaps) and its own internal spins of a coin that result in it saying 1 a proportion p of the time, and 0 a proportion $1-p$ of the time. Both those are at least statistically *independent* of the truth of if the bit is 1 or 0, as the encryption is secure in the first case and because of causal independence in the second case, which justifies the calculation (*).

That is semantic security at runtime for object code from an 'obfuscating compiler',[7] following Theorem 2, modulo the assumption that encryption is secure and conditions (1-4) hold. Has data obfuscation as defined in Section 1 been obtained? Yes. The flat distribution of possible data values under the encryption means no information can be gained from traces.

CONCLUSION

This paper has considered privacy and security of data on platforms for encrypted computing against the operator or operating system as an adversary, assuming the encryption is secure in its own right.

Conditions on the processor and machine code architecture have been defined such that a compiler may obfuscate the runtime data under the encryption, producing uniformly distributed variations across different compilations, at every point in the program. That eliminates attacks based on the use by a human author of small numbers in program or data. No unencrypted data value can then be statistically inferred from code and trace, making a known plaintext attack on the encryption impossible. That also amounts to semantic security of an integrated system for encrypted computing consisting of a processor with an instruction set satisfying the conditions set out, plus an 'obfuscating compiler', modulo the security of the encryption.

[7] Haskell source code for a prototype obfuscating C compiler following our design may be downloaded from nbd.it.uc3m.es/∼ptb/obfusc_comp-0_9a.hs. The compiler produces generic 'fused operate and add' instructions.

ACKNOWLEDGEMENTS

Zhiming Liu wishes to thank the Chinese NSF for support from research grant 61672435, and Southwest University for research grant SWU116007. Peter Breuer wishes to thank Hecusys LLC (hecusys.com) for continued support in KPU development.

REFERENCES

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (im)possibility of obfuscating programs. In Kilian, J., editor, *Proc. 21st Annu. Int. Cryptol. Conf. (CRYPTO'01)*, Adv. Cryptol., pages 1–18. Springer.

Breuer, P. T. and Bowen, J. P. (2014). Towards a working fully homomorphic crypto-processor: Practice and the secret computer. In Jörjens, J., Pressens, F., and Bielova, N., editors, *Proc. Int. Symp. Eng. Sec. Softw. Syst. (ESSoS'14)*, volume 8364 of *LNCS*, pages 131–140, Berlin/Heidelberg. Springer.

Breuer, P. T. and Bowen, J. P. (2016). A fully encrypted microprocessor: The secret computer is nearly here. *Procedia Comp. Sci.*, 83:1282–1287.

Breuer, P. T., Bowen, J. P., Palomar, E., and Liu, Z. (2016). A practical encrypted microprocessor. In Callegari, C., van Sinderen, M., Sarigiannidis, P., Samarati, P., Cabello, E., Lorenz, P., and Obaidat, M. S., editors, *Proc. 13th Int. Conf. Sec. Cryptog. (SECRYPT'16)*, volume 4, pages 239–250, Portugal. SCITEPRESS.

Conway, J. H. (1987). Fractran: A simple universal programming language for arithmetic. In *Open Problems in Commun. & Comput.*, pages 4–26. Springer.

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

Fletcher, C. W., van Dijk, M., and Devadas, S. (2012). A secure processor architecture for encrypted computation on untrusted programs. In *Proc. 7th Scal. Trust. Comput. Workshop (STC'12)*, pages 3–8, NY. ACM.

Hada, S. (2000). Zero-knowledge and code obfuscation. In Okamoto, T., editor, *Proc. 6th Int. Conf. Theor. Applicat. Cryptol. Inform. Sec. (ASIACRYPT'00)*, number 1976 in LNCS, pages 443–457. Springer.

Ostrovsky, R. and Goldreich, O. (1992). Comprehensive software protection system. US Pat. 5,123,045.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT'99*, Adv. Cryptol., pages 223–238. Springer.

Patterson, D. (1985). Reduced instruction set computers. *Commun. ACM*, 28(1):8–21.

Tsoutsos, N. and Maniatakos, M. (2015). The HEROIC framework: Encrypted computation without shared keys. *IEEE Trans. CAD IC Syst.*, 34(6):875–888.

van Dijk, M. and Juels, A. (2010). On the impossibility of cryptography alone for privacy-preserving cloud computing. *HotSec*, 10:1–8.

Wang, Z. and Lee, R. B. (2006). Covert and side channels due to processor architecture. In *Proc. 2nd Annu. Comp. Sec. Applic. Conf. (ACSAC'06)*, pages 473–482. IEEE.

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. (2013). Practical control flow integrity and randomization for binary executables. In *Symp. Sec. Priv.*, pages 559–573. IEEE.