An Information Obfuscation Calculus for Encrypted Computing

Peter T. Breuer
Hecusys LLC, Atlanta, GA, USA
Email: ptb@hecusys.com

***Abstract*—Relative cryptographic semantic security for encrypted words of user data at runtime holds in the emerging field of encrypted computing, in conjunction with an appropriate instruction set and compiler. An information obfuscation calculus for program compilation in that context is introduced here that quantifies the security exactly, improving markedly on the result.**

I. Introduction

This article describes 'obfuscating' compilation for the emerging field of *encrypted computing* [1] via a calculus that quantifies the obfuscation. Encrypted computing means running on a processor that works profoundly encrypted for an unprivileged user process, taking encrypted inputs to encrypted outputs via encrypted intermediate values in registers and memory. Our prototype compiler (http://sf.net/p/obfusc) is for ANSI C [2] but the approach is generic, guided by the information principle:

> *Every machine code arithmetic instruction that writes must introduce maximal possible entropy to the trace.* ($\widetilde{\mathbb{H}}$)

A target processor works unencrypted in operator mode in the conventional way, with unrestricted access to registers and memory. User mode is restricted to certain registers and memory. Encryption keys are not accessible to the operator and operating system, who are the user's (potential) adversaries in this context. Keys are installed at manufacture, as in Smartcards [3], or uploaded securely in public view via a Diffie-Hellman circuit [4]. Hardware questions, such as the real randomness of random numbers or power side-channel information leaks, are not an issue here.

The text will use 'the operator' for operator mode. A malicious operating system is the operator, as is a human with administrative privileges, perhaps obtained by physically interfering with the boot process. A possible context for an attack is where user data consists of scenes from animation cinematography being rendered in a server farm. The computer operators at the server farm have an opportunity to pirate for profit portions of the movie before release and they may be tempted. Another scenario is a specialised facility processing satellite photos of a foreign power's military installations to reveal changes since a previous pass. If an operator (or a hacked operating system) can modify the data to show no change where there has been some, then that is an opportunity for espionage. A *successful attack* by the operator is one that discovers the plaintext of user data or alters it to order.

It is shown in [5] that (i) a processor that supports encrypted computing, (ii) an appropriate machine code instruction set architecture, (iii) a compiler with an 'obfuscating' property, jointly give *cryptographic semantic security* (CSS) [6] for user data against the operator as adversary, relative to the security of the encryption. That is, the operator cannot read any bit of user data beneath the encryption with probability of success above ½. [5] shows that also implies data cannot be rewritten to an independently defined value such as π, or the encryption key.

'Obfuscating' compilation in this context is described in [7]. It varies object codes on recompilation so (a) they look the same to an adversary, differing only in encrypted constants, which the adversary cannot read. Also (b) runtime traces 'look the same' too, with the same instructions in the same order reading and writing the same registers, while data beneath the encryption varies by an arbitrary delta different at every point in the runtime trace and memory, with the proviso that:

> *Copy instructions preserve data exactly, and deltas are equal where control paths meet.* ($\widetilde{\mathbb{H}}$)

That is necessary in order for computation to work properly. Loops in particular have the same delta from nominal beneath the encryption at either end, ready for a repeat traversal.

The security proofs in [5] rely on object codes and runtime traces varying as described above. This paper describes 'correct by construction' compilation for that property following the principle ($\widetilde{\mathbb{H}}$). For the compiled programs, at any m points in the trace not related as in ($\widetilde{\mathbb{H}}$), it is proved that variations with $32m$ bits of entropy beneath the encryption occur. That quantifies the security in encrypted computing exactly, improving the existing CSS result.

This paper is organised as follows. Section II describes encrypted computing platforms. Section III resumes a concrete, modified OpenRISC (http://openrisc.io) machine code instruction set for encrypted computing first described in [7] and satisfying the properties required (Box 1). The obfuscating compiler technology of [7] is resumed in IV-A and IV-B. A pre-/post-condition Hoare program logic [8] for the offset calculus that the compiler uses to keep track of its code variations is introduced in IV-C, and in IV-D it is modified to an obfuscation calculus for compiler-induced entropy.

Notation

Encryption is denoted by $x^{\mathcal{E}}{=}\mathcal{E}[x]$ of plaintext value x. The operation on the ciphertext domain corresponding to o on the plaintext domain is written $[o]$, where $\mathcal{E}[x]\ [o]\ \mathcal{E}[y]{=}\mathcal{E}[x\,o\,y]$. Encryption $x^{\mathcal{E}}$ appears many-valued, due to unobservable padding alongside x, so '$=$' above means equivalence modulo those variations.

II. Background

Several fast processors for encrypted computing are described in [9]. Those include the KPU [10], which runs encrypted on

Box 1: Machine code instruction set axioms. Instructions ...

(1) *are a black box from the perspective of the programming interface, with no internal states visible;*
(2) *take encrypted inputs to encrypted outputs;*
(3) *are adjustable via (encrypted) embedded constants to any offsets in decrypted inputs and outputs;*
(4) *are such that there are no collisions between encrypted constants and runtime encrypted values.*

Box 2: What the compiler does:

(A) *change only encrypted program constants, generating an arrangement of planned* offsets from nominal values *for instruction inputs and outputs beneath the encryption* (an obfuscation scheme);
(B) *leave runtime traces unchanged, apart from differences in the encrypted program constants* (A) and *runtime data*;
(C) *equiprobably generate all arrangements satisfying* (A), (B).

a 1 GHz clock with AES-128 [11] at the benchmark speed of a 433 MHz classic Pentium, and the slightly older HEROIC [12] which runs like a 25 KHz Pentium, embedding Paillier-2048 [13], as well as the recently announced CryptoBlaze [14], also using Paillier-2048 but 10× faster than HEROIC (it is not clear how many of those have working compilers).

The machine code instruction set defining the programming interface is important because a conventional instruction set is insecure against powerful insiders who may steal an (encrypted) user datum x and put it through the machine's division instruction to get x/x encrypted, an encrypted 1. Then any desired encrypted y may be constructed by applying the machine's addition instruction to get $1+\ldots+1$ encrypted. Via the order comparator instructions (testing $2^{31}\leq z, 2^{30}\leq z, \ldots$) on an encrypted z and subtracting on branch, z may be obtained bitwise. That is a *chosen instruction attack* [5], [15]. An instruction set for encrypted computing must resist *algebraic* attacks like that, but the compiler must also be involved, else there would still be *known plaintext attacks* [16] based on the idea that human programmers intrinsically use values like 0, 1 more often than others. The compiler's job is to even out the statistics.

Necessary conditions on the machine code instruction set, first described in [7], are shown in Box 1. Instructions must (1) execute atomically, or recent attacks such as Meltdown [17] and Spectre [18] against Intel are feasible, (2) work with encrypted values or an adversary could read them, and (3) be adjustable via onboard (encrypted) constants to offset by arbitrary deltas the runtime values beneath the encryption. The condition (4) is for the security proofs in [7], and amounts to having different padding or blinding factors for encrypted program constants and encrypted runtime values.

The compiler's job is to vary the encrypted constants (3) embedded in the machine code instructions so all feasible trace variations are exercised *equiprobably*. 'How' is summarised in Box 2: a new *obfuscation scheme* is generated at each recompilation. That is a set of vectors of *planned offsets from nominal for the data beneath the encryption per memory and register location, one vector at each machine code instruction*.

A formal outline is as follows. The compiler $\mathbb{C}[-]$ translates an expression e that is to end up in register r at runtime to machine code mc and plans a 32-bit offset Δe in r:

$$\mathbb{C}[e]^r = (mc, \Delta e) \tag{5}$$

Let $s(r)$ be the value in register r in state s of the processor at runtime. The machine code *mc* changes state s to s' that holds a ciphertext in r whose plaintext value differs by Δe from the nominal value e (bitwise exclusive-or or the binary operation of a mathematical group are alternatives for '+' here). That is:

$$s \overset{mc}{\rightsquigarrow} s' \text{ where } s'(r) = \mathcal{E}[e + \Delta e] \tag{6}$$

The encryption $\mathcal{E}$ is shared with the user and the processor, but not operator and operating system. The randomly generated compiler offsets Δe are known to the user, but not the processor nor operator and operating system. The user compiles the program and sends it to the processor to be executed and knows the obfuscation scheme, so can create the right inputs and interpret the outputs received.

TABLE I
FXA INSTRUCTION SET FOR ENCRYPTED WORK.

op.	*fields*	*mnem.*	*semantics*
add	$\mathrm{r}_0\,\mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	add	$r_0 \leftarrow r_1\ [+]\ r_2\ [+]\ k^{\mathcal{E}}$
sub	$\mathrm{r}_0\,\mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	subtract	$r_0 \leftarrow r_1\ [-]\ r_2\ [+]\ k^{\mathcal{E}}$
mul	$\mathrm{r}_0\,\mathrm{r}_1\,\mathrm{r}_2\,k_0^{\mathcal{E}}\,k_1^{\mathcal{E}}\,k_2^{\mathcal{E}}$	multiply	$r_0 \leftarrow (r_1\ [-]\ k_1^{\mathcal{E}})\ [*]\ (r_2\ [-]\ k_2^{\mathcal{E}})\ [+]\ k_0^{\mathcal{E}}$
div	$\mathrm{r}_0\,\mathrm{r}_1\,\mathrm{r}_2\,k_0^{\mathcal{E}}\,k_1^{\mathcal{E}}\,k_2^{\mathcal{E}}$	divide	$r_0 \leftarrow (r_1\ [-]\ k_1^{\mathcal{E}})\ [\div]\ (r_2\ [-]\ k_2^{\mathcal{E}})\ [+]\ k_0^{\mathcal{E}}$
...			
mov	$\mathrm{r}_0\,\mathrm{r}_1$	move	$r_0 \leftarrow r_1$
beq	$i\ \mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1\ [=]\ r_2\ [+]\ k^{\mathcal{E}}$
bne	$i\ \mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1\ [\neq]\ r_2\ [+]\ k^{\mathcal{E}}$
blt	$i\ \mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1\ [<]\ r_2\ [+]\ k^{\mathcal{E}}$
bgt	$i\ \mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1\ [>]\ r_2\ [+]\ k^{\mathcal{E}}$
ble	$i\ \mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1\ [\leq]\ r_2\ [+]\ k^{\mathcal{E}}$
bge	$i\ \mathrm{r}_1\,\mathrm{r}_2\,k^{\mathcal{E}}$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1\ [\geq]\ r_2\ [+]\ k^{\mathcal{E}}$
...			
b	i	branch	$pc \leftarrow pc + i$
sw	$(k_0^{\mathcal{E}})\mathrm{r}_0\ \mathrm{r}_1$	store	$\mathrm{mem}[\![r_0\ [+]\ k_0^{\mathcal{E}}]\!] \leftarrow r_1$
lw	$\mathrm{r}_0\ (k_1^{\mathcal{E}})\mathrm{r}_1$	load	$r_0 \leftarrow \mathrm{mem}[\![r_1\ [+]\ k_1^{\mathcal{E}}]\!]$
jr	r	jump	$pc \leftarrow r$
jal	j	jump	$ra \leftarrow pc + 4;\ pc \leftarrow j$
j	j	jump	$pc \leftarrow j$
nop		no-op	

LEGEND

r – register index	k – 32-bit integer	pc – prog. count reg.
j – prog. count	'$\leftarrow$' – assignment	ra – return addr. reg.
$\mathcal{E}[\ ]$ – encryption	i – prog. incr.	r – register content
$k^{\mathcal{E}}$ – encrypted val. $\mathcal{E}[k]$	$x^{\mathcal{E}}\ [o]\ y^{\mathcal{E}} = \mathcal{E}[x\ o\ y]$	$x^{\mathcal{E}}\ [R]\ y^{\mathcal{E}} \Leftrightarrow x\ R\ y$

III. FXA INSTRUCTIONS

A 'fused anything and add' (FxA) [7] instruction set architecture (ISA) will be the compilation target here, satisfying (1-4) above. The integer portion is shown in Table I. The whole is adapted from the OpenRISC ISA v1.1 (http://openrisc.io/or1k.html), which has about 200 instructions. There are instructions for single and double precision integer operations, single and double floating point, and vector operations, all 32 bits long. Following OpenRISC, instructions access up to three 32 general purpose registers (GPRs), but one register operand may be replaced by a ('immediate') constant. Four 32-bit 'prefixes' precede a 32-bit instruction with 16 bits of room to provide $128{=}4{\times}28{+}16$ bits for one encrypted constant.

IV. Obfuscating Compilation

A compiler works with a database $D : \mathrm{Loc} \to \mathrm{Off}$ containing (here 32-bit) integer 'offset deltas' Δl for data, indexed per register or memory location l (type Loc). It is varied by the compiler as it makes its pass through the source code. A delta defines by how much the runtime data underneath the encryption is to differ from nominal in l, and the database D at each point constitutes an *obfuscation scheme of offsets*.

A database $L : \mathrm{Var} \to \mathrm{Loc}$ conventionally maps source code variables to registers and memory and will not be treated here.

In [7], an expression compiler that places the encrypted result value in target register r is described, of type:

$$\mathbb{C}^L[\_ : \_]^r : \mathrm{DB} \times \mathrm{Expr} \to \mathrm{MC} \times \mathrm{Off} \tag{7}$$

where MC stands for machine code, a sequence of FxA instructions *mc*. The compiler aims to vary the Δl equiprobably across recompilations. The next section resumes how it does it.

A. Expressions

To translate $x + y$ where x, y are signed 32-bit integer variables, the compiler first emits machine code mc_0 as in (8a). At runtime that will put x in register r_0 with offset delta Δx (a pair in DB×Expr is written $D : x$ for readability):

$$(mc_0, \Delta x) = \mathbb{C}^L[D : x]^{r_0} \tag{8a}$$

$$s_0 \overset{mc_0}{\rightsquigarrow} s_1 \;:\; s_1(r_0) = \mathcal{E}[s_0(Lx) + \Delta x] \tag{8b}$$

The semantics for (8b) is from Table I with $s(r)$ in register r.

The compiler next emits machine code mc_1 (9a). At runtime that will put y in register r_1 with offset delta Δy:

$$(mc_1, \Delta y) = \mathbb{C}^L[D : y]^{r_1} \tag{9a}$$

$$s_1 \overset{mc_1}{\rightsquigarrow} s_2 \;:\; s_2(r_1) = \mathcal{E}[s_1(Ly) + \Delta y] \tag{9b}$$

An offset Δe is randomly generated and the compiler emits the FxA integer **add** instruction that at runtime adds the sum from r_0 and r_1 in r_0, modified by a delta k:

$$\mathbb{C}^L[D : x + y]^{r_0} = (mc_e, \Delta e) \tag{10a}$$
$$mc_e = mc_0; mc_1; \mathbf{add}\ r_0\ r_0\ r_1\ k^{\mathcal{E}}$$

Choosing $k{=}\Delta e{-}\Delta x{-}\Delta y$, the expression gets offset Δe:

$$s_0 \overset{mc_e}{\rightsquigarrow} s_2 \;:\; s_2(r_0) = \mathcal{E}[s_0(Lx) + s_1(Ly) + \Delta e] \tag{10b}$$

Any Δe may be set, no matter what Δx, Δy were chosen.

B. Statements

Let Stat be the type of statements, then compiling a statement produces a new obfuscation scheme:

$$\mathbb{C}^L[\_ : \_] : \mathrm{DB} \times \mathrm{Stat} \to \mathrm{DB} \times \mathrm{MC} \tag{11}$$

Consider an assignment $z{=}x{+}y$ of expression $x{+}y$ to a source code variable z, which the location database L binds in register $r_z{=}Lz$. Let $x{+}y$ be called e here. The compiler emits code mc_e that evaluates expression e in register **t0** with randomly generated offset Δe as described in (10a) with $\mathbf{t0} = r_0$. A short-form **add** instruction with semantics $r_z \leftarrow \mathbf{t0}\,[+]\,k^{\mathcal{E}}$ is emitted to change Δe to Δz:

$$\mathbb{C}^L[D_0 : z{=}e] = D_1 : mc_e; \mathbf{add}\ r_z\ \mathbf{t0}\ k^{\mathcal{E}} \tag{12a}$$

The compiler sets $k{=}\Delta z{-}\Delta e$ to choose Δz for z in $r_z = Lz$:

$$s_0 \overset{mc_e}{\rightsquigarrow} s_2 \overset{\mathrm{add}}{\rightsquigarrow} s_3 \;:\; s_3(r_z) = \mathcal{E}[s_0(Lx) + s_1(Ly) + \Delta z] \tag{12b}$$

The database of offsets is updated from $D_0 r_z$ to $D_1 r_z{=}\Delta z$.

C. Offset Calculus

A classical pre-/post-condition calculus [8] is given below that captures the compiler's changes above to the obfuscation scheme.

1) Assignment: Generalising the $x{+}y$ above to expression e with intermediates in registers $\rho{=}\{r_0, \ldots, r_n\}$, and result z in r_0, the offsets before and after the assignment are generically:

$$\begin{gathered}\{\Delta r_0 = Y_0,\ \ldots,\ \Delta r_n = Y_n\} \\ z = e \\ \{\Delta' r_0{=}\ Z_0,\ \ldots,\ \Delta' r_n{=}\ Z_n\}\end{gathered} \tag{13}$$

By the example (10b,12b), the Δ', Δ are independently chosen as the compiler modifies (post-) scheme Δ' to (pre-) scheme Δ:

$$\{\Delta\}\ z = e\ \{\Delta'\} \tag{13a}$$
$$\text{where } \Delta \supseteq \Delta'|_{\bar\rho} \tag{13b}$$

and Δ, Δ' are identical on the *complement* $\bar\rho$ of ρ.

2) Conditionals: Source code conditionals are compiled to machine code branch instructions, but which branch is for true and which for false from expression e is determined by an encrypted bit in the machine code, so it cannot be read by an adversary and the pre-/post- logic is that of classic nondeterministic choice. Let ρ be the registers written in e. The deduction is:

$$\frac{\{\Delta_1\}\ s_1\ \{\Delta'\} \quad \{\Delta_2\}\ s_2\ \{\Delta'\}}{\{\Delta\}\ \mathbf{if}\ (e)\ s_1\ \mathbf{else}\ s_2\ \{\Delta'\}} \tag{14a}$$
$$\Delta \supseteq \Delta_1|_{\bar\rho} \cup \Delta_2|_{\bar\rho} \tag{14b}$$

and Δ, Δ_1, Δ_2 are identical on $\bar\rho$, otherwise independent. The final offsets Δ' set by the compiler are equal in both branches, as following code must be parameterised on specific offsets.

3) Loops: The compiler implements **do while** loops as body plus conditional branch back to the start. Let ρ be the registers written in e. The other registers must be offset equally at loop start and end, i.e., $\Delta_1|_{\bar\rho}{=}\Delta_2|_{\bar\rho}{=}\Delta'|\bar\rho$ in (14a),(14b):

$$\frac{\{\Delta\}\ s\ \{\Delta'\}}{\{\Delta\}\ \mathbf{do}\ s\ \mathbf{while}\ e\ \{\Delta'\}} \tag{15a}$$
$$\Delta \supseteq \Delta'|_{\bar\rho} \tag{15b}$$

The compiler is free to set offsets $\Delta|_\rho$ and $\Delta'|\rho$ independently.

D. Obfuscation Calculus

Let f_r be the probability distribution of offset Δr from a nominal value v beneath the encryption in register r, so $\mathrm{prob}(s(r) = \mathcal{E}[v{+}d]){=}\mathrm{prob}(\Delta r{=}d){=}f_r(d)$, where s is the processor state.

Each Δr, $\Delta' r$ is a random variable with a probability

distribution, giving the stochastic analogue of (13) below:

$$\{\Delta r_x = \mathcal{X},\ \Delta r_y = \mathcal{Y},\ \Delta r_z = \mathcal{Z}\}$$
$$z = x + y$$
$$\{\Delta' r_x = \mathcal{X},\ \Delta' r_y = \mathcal{Y},\ \Delta' r_z = \mathcal{Z}'\}$$

In particular, Δr_z and $\Delta' r_z$ are independent random variables.

Let T be the runtime trace of a program. It is a linear listing of each instruction executed and values it read and wrote. After an assignment the trace is longer by one: $T' = T^\frown\langle z = e\rangle$.

The entropy $\mathrm{H}(T)$ of the random variable T distributed as f_T is the expectation $\mathbb{E}[-\log_2 f_T]$, and the increase in entropy from T to T' (it cannot decrease as T lengthens) is the number of bits of unpredictable information added. The flat distribution $f_T{=}k$ has maximal entropy $\mathrm{H}(T){=}\log_2(1/k)$. Adding a maximal entropy signal to any random variable on a n-bit space gives another maximal entropy, i.e., flat, distribution.

1) Assignment: As in (13a), for pre-/post-condition:

$$\{\Delta\}\ z = e\ \{\Delta'\} \tag{16a}$$

but the bindings Δ, Δ' are of offsets Δr, $\Delta' r$ that are random variables. Let $\rho{=}\{r_0, \ldots, r_n\}$ be the registers written in e or in writing to z. For $r{\notin}\ \rho$, $\Delta' r = \Delta r$, because they are equal values by (13b), so the precondition is also $\Delta|_{\bar\rho}{=}\Delta'|_{\bar\rho}$ here. I.e.:

$$\Delta \supseteq \Delta'|_{\bar\rho} \tag{16b}$$

but there is more, because ($\widetilde{\mathbb{H}}$) means each new random variable is independent with maximal entropy. Each represents the compiler's free choice of (encrypted) constant in 'an arithmetic instruction that writes', as remarked following (13),(14b),(15b).

Let the trace entropy up to the assignment be $\mathrm{H}(T){=}h$. Writing with offset a new independent r.v. U increases it to $\mathrm{H}(T'){=}h{+}\mathrm{H}(U)$. The offset is 32-bit, chosen with flat distribution by the compiler for ($\widetilde{\mathbb{H}}$), so $\mathrm{H}(U){=}32$, and $n{+}1$ registers $r_0, \ldots, r_n$ are written so entropy increases by $32(n{+}1)$ bits:

$$\{\mathrm{H}(T) + 32(n{+}1) = h\}\ \ z = e\ \ \{\mathrm{H}(T') = h\} \tag{16c}$$

But where the instruction has already appeared in the trace, the offset is pre-known, and the increment is zero:

$$\{\mathrm{H}(T) = h\}\ \ z = e\ \ \{\mathrm{H}(T') = h\} \tag{16c0}$$

2) Conditionals: As in (13b),(14b) but with random variables:

$$\frac{\{\Delta_1\}\ s_1\ \{\Delta'\}\quad \{\Delta_2\}\ s_2\ \{\Delta'\}}{\{\Delta\}\ \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2\ \{\Delta'\}} \tag{17a}$$
$$\Delta \supseteq \Delta_1|_{\bar\rho} \cup \Delta_2|_{\bar\rho} \tag{17b}$$

The $\Delta r{=}\Delta_1 r{=}\Delta_2 r$ for $r{\notin}\rho$, because they are equal values according to (14b). The entropy added to the trace T is from the trace of e, plus that from the trace through a branch:

$$\frac{\{\mathrm{H}(T){=}h{+}32n\}\ s_1\ \{Q\}\quad \{\mathrm{H}(T){=}h{+}32n\}\ s_2\ \{Q\}}{\{\mathrm{H}(T) = h\}\ \ \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2\ \ \{Q\}} \tag{17c}$$

To make that deduction valid, the compiler must even up the arithmetic writes between the two branches so the entropy increase is the same. It can do it, because, even for loops, the entropy increase is finite and bounded, as discussed below.

A second time the conditional appears in the trace, if it branches the same way again then it contributes zero entropy as all the offset deltas are already known:

$$\{\mathrm{H}(T) = h\}\ \ \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2\ \ \{\mathrm{H}(T') = h\} \tag{17c0}$$

If it branches a different way, the branch (but not the test) contributes entropy, as the offsets in that branch are yet unknown. But the, say m, instructions that align final register offsets are constrained in (14b). So those m do not count:

$$\frac{\{\mathrm{H}(T){=}h\}\ s_1\ \{Q\}\ \{\mathrm{H}(T){=}h\}\ s_2\ \{Q\}}{\{\mathrm{H}(T){+}32m = h\}\ \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2\ \{Q\}} \tag{17c1}$$

Definition 1: An instruction emitted to adjust the final offset to a common value with the other branch is a *trailer* instruction. Each is last to write to a register in the branch.

3) Loops: Let $\rho{=}\{r_1, \ldots, r_n\}$ be the registers written in e. Then, per (15a), (15b), but with random variables:

$$\frac{\{\Delta\}\ s\ \{\Delta'\}}{\{\Delta\}\ \textbf{do}\ s\ \textbf{while}\ (e)\ \{\Delta'\}} \tag{18a}$$
$$\Delta \supseteq \Delta'|_{\bar\rho} \tag{18b}$$

That means $\Delta r{=}\Delta' r$ for $r{\notin}\rho$. The distributions are equal because the values are equal for $r{\notin}\rho$, by (16b).

A trace over the loop is always the same length, because the compiler varies data values, not semantics. Say the loop repeats $N{\geq}1$ times for a particular set of input values. Then it could be unrolled to N instances of the loop body and N instances of the loop test. The variation in the trace is only that of (a) the test repeated once, because the same offsets are applied to the n registers that are written in e at each repeat, plus (b) that of the body repeated once, for the same reason. The entropy calculation is (a) plus (b), no matter what N is:

$$\frac{\{\mathrm{H}(T){+}32m = h\}\ s\ \{\mathrm{H}(T') = h\}}{\{\mathrm{H}(T){+}32(n{+}m){=}h\}\ \textbf{do}\ s\ \textbf{while}\ e\ \{\mathrm{H}(T'){=}h\}} \tag{18c}$$

The abstraction of a **do while** loop here is that it lengthens the trace arbitrarily but adds entropy like a conditional.

On a second time through the loop, zero entropy is added, because the offsets are the same as the last time:

$$\{\mathrm{H}(T) = h\}\ \ \textbf{do}\ s\ \textbf{while}\ e\ \ \{\mathrm{H}(T) = h\} \tag{18c0}$$

The (red) equations are an obfuscation calculus for trace entropy when compilation follows the principle ($\widetilde{\mathbb{H}}$). In summary:

Lemma 1: The entropy of a trace is $32(n{+}i)$ bits; n is the number in it of distinct arithmetic instructions that write (a pair of trailer instructions count as the same) and i is the number of inputs.

Inputs may be counted as those instructions that read first time a location that has not been yet been written in the trace.

The compiler must recruit every arithmetic instruction that writes to the task of freely varying the offsets in data beneath the encryption in register and memory locations. The sole restriction is that two final writes in different control paths must set up the same offsets, and that is for correct program working. Conditionals, loops and gotos could go wrong otherwise.

Proposition 1: The entropy of a program trace is maximal

w.r.t. varying the constant parameters in the compiled machine code, while it still works correctly in any context.

Observing data at a point in the trace that has been written by a program instruction (or read from a location in memory that has not yet been written) sees variation across recompilations. 'Any context' allows that data written may always be read so synchronising final offsets between branches is necessary. If the data is never read, synchronising could be done without, as nothing depends on it. The proposition incidentally implies a full 32 bits of entropy per datum are provided by the compiler:

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

That result was obtained by structural induction in [7].

Definition 2: Two data observations in the trace are (obfuscation) *dependent* if they are of the same register at the same point, are input and output of a copy instruction, or are of the same register near a join of two control paths after the last write to it in each and before the next write.

If the trace is observed at two (in general, m) independent points, the variation is maximal:

Theorem 1: The probability across different compilations that any m particular 32-bit values have encryptions $\mathcal{E}[x_i]$ in given register or memory location at given points in the program at runtime, provided they are pairwise independent, is $1/2^{32m}$.

Each dependent pair reduces the entropy by 32 bits. That provides a proof for the result in [5] as every write or read taken in isolation injects or inherits 32 bits of variability:

Corollary 2: Runtime user data beneath the encryption is *cryptographically semantically secure* [6] against the operator for code compiled by the obfuscating compiler.

That property means any attack is 'no better than guessing' for any one bit in isolation. Theorem 1 asserts the stronger result that as many trace data words under the encryption as one cares to look at simultaneously are maximally unpredictable across recompilations, as far as is possible.

V. Implementation

Our own prototype compiler http://sf.net/p/obfusc following IV-D is for ANSI C [2], where pointers and arrays present particular difficulties. Currently, the compiler has near total coverage of ANSI C and GNU C extensions, including statements-as-expressions and expressions-as-statements, gotos, arrays, pointers, structs, unions, floating point, double integer and floating point data. Pointers are obligatorily declared via ANSI **restrict** to point into arrays. It is missing **longjmp** and efficient strings (**char** and **short** are same as **int**), and global data shared across code units (a linker issue). The largest C source compiled (correctly) so far is 22,000 lines for the IEEE floating point test suite at http://jhauser.us/arithmetic/TestFloat.html. A trace[1] of the Ackermann function[2] [19] is shown in Table II.

[1]For readability here, the final offset delta for register **v0** is set to zero.

[2]Ackermann C code: **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)); }.

TABLE II
TRACE FOR ACKERMANN(3,1), RESULT 13.

PC	instruction	update trace
...		
35	add t0 a0 zer $\mathcal{E}$[-86921031]	t0 = $\mathcal{E}$[-86921028]
36	add t1 zer zer $\mathcal{E}$[-327157853]	t1 = $\mathcal{E}$[-327157853]
37	beq t0 t1 2 $\mathcal{E}$[240236822]	
38	add t0 zer zer $\mathcal{E}$[-1242455113]	t0 = $\mathcal{E}$[-1242455113]
39	b 1	
41	add t1 zer zer $\mathcal{E}$[-1902505258]	t1 = $\mathcal{E}$[-1902505258]
42	xor t0 t0 t1 $\mathcal{E}$[-1734761313]	$\mathcal{E}$[1242455113] $\mathcal{E}$[1902505258]
		t0 = $\mathcal{E}$[-17347613130]
43	beq t0 zer 9 $\mathcal{E}$[-1734761313]	
53	add sp sp zer $\mathcal{E}$[800875856]	sp = $\mathcal{E}$[1687471183]
54	add t0 a1 zer $\mathcal{E}$[-915514235]	t0 = $\mathcal{E}$[-915514234]
55	add t1 zer zer $\mathcal{E}$[-1175411995]	t1 = $\mathcal{E}$[-1175411995]
56	beq t0 t1 2 $\mathcal{E}$[259897760]	
57	add t0 zer zer $\mathcal{E}$[11161509]	t0 = $\mathcal{E}$[11161509]
...		
143	add v0 t0 zer $\mathcal{E}$[42611675]	v0 = $\mathcal{E}$[13]
...		
147	jr ra	# (return $\mathcal{E}$[13] in *v0*)

Legend: (registers) a0 = function argument; sp = stack pointer; t0, t1 = temporaries; v0 = return value; zer = null placeholder.

VI. Conclusion

A formal obfuscation calculus for programs has been set out that calculates the entropy in the data beneath the encryption in a runtime trace on an encrypted computing platfrom, where compilation follows the principle that every arithmetic instruction that writes is varied maximally across recompilations.

A stronger, quantified, cryptographic semantic security property follows for encrypted computing: no attack has better than probability $1/2^{32m}$ of success on a trace that contains m 'independent' words of user data, relative to the security of the encryption.

References

[1] P. Breuer and J. Bowen, "A fully homomorphic crypto-processor design: Correctness of a secret computer," in *Proc. Int. Symp. Eng. Sec. Softw. Sys. (ESSoS'13)*, ser. LNCS. Springer, 2013, no. 7781, pp. 123–38.

[2] ISO/IEC, "Programming languages – C," Int. Org. for Standardization, 9899:201x Tech. Rep. n1570, Aug. 2011, JTC 1, SC 22, WG 14.

[3] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Proc. USENIX Work. Smartcard Tech.* USENIX, 1999, pp. 9–20.

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

[5] P. Breuer, J. Bowen, E. Palomar, and Z. Liu, "On security in encrypted computing," in *Proc. 20th Int. Conf. Info. Comm. Sec. (ICICS'18)*, ser. LNCS, D. Naccache *et al.*, Eds. Springer, 2018, no. 11149, pp. 192–211.

[6] S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *Proc. 14th Ann. ACM Symp. Th. Comp.*, ser. (STOC'82). ACM, 1982, pp. 365–77.

[7] P. Breuer, J. Bowen, E. Palomar, and Z. Liu, "On obfuscating compilation for encrypted computing," in *Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17)*, P. Samarati *et al.*, Eds. SCITEPRESS, 2017, pp. 247–54.

[8] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–80, 1969.

[9] P. Breuer, J. Bowen, E. Palomar, and Z. Liu, "Superscalar encrypted RISC: The measure of a secret computer," in *Proc. 17th Int. Conf. Trust, Sec. & Priv. in Comp. & Comms. (TrustCom'18)*. IEEE Comp. Soc., 2018, pp. 1336–41.

[10] ——, "A practical encrypted microprocessor," in *Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16)*, C. Callegari *et al.*, Eds., vol. 4. SCITEPRESS, 2016, pp. 239–50.

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

[12] N. G. Tsoutsos and M. Maniatakos, "The HEROIC framework: Encrypted computation without shared keys," *IEEE TCAD IC Sys.*, vol. 34, no. 6, pp. 875–88, 2015.

[13] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Th. Appl. Crypto. Tech. (EUROCRYPT'99)*, ser. LNCS, J. Stern, Ed., no. 1592. Springer, 1999, pp. 223–38.

[14] F. Irena, D. Murphy, and S. Parameswaran, "Cryptoblaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support," in *Proc. 23rd Asia S. Pac. Des. Autom. Conf. (ASP-DAC'18)*. IEEE, 2018, pp. 702–8.
[15] S. Rass and P. Schartner, "On the security of a universal cryptocomputer: The chosen instruction attack," *IEEE Access*, vol. 4, pp. 7874–82, 2016.
[16] A. Biryukov, "Known plaintext attack," in *Ency. Cryptog. & Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 704–5.
[17] M. Lipp *et al.*, "Meltdown," *ArXiv e-prints*, Jan. 2018.
[18] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.
[19] Y. Sundblad, "The Ackermann function: a theoretical, computational, and formula manipulative study," *BIT Num. Math.*, vol. 11, no. 1, pp. 107–19, 1971.