EasyChair Preprint

№ 2049

An Information Entropy Calculus for Stochastically Compiled Programs

Peter T. Breuer and Simon J. Pickin

November 29, 2019

An Information Entropy Calculus for Chaotically Compiled Programs

Peter T. Breuer[1] and Simon J. Pickin[2]

[1] Hecusys LLC, Atlanta, GA, USA. ptb@hecusys.com
[2] Universidad Complutense, Madrid, Spain. simon.pickin@fdi.ucm.es

Abstract. A calculus for the entropy in runtime traces resulting from stochastic program compilation is introduced here. It quantifies the variation that a 'chaotic' compiler aiming to vary the object code introduces into the program trace at run time, and the best strategy is characterised.

1 Introduction

This article describes stochastic compilation via a program calculus that quantifies the notion. To be clear from the outset, the stochastic element occurs not in the execution but in the compilation of a program. Our prototype HAVOC compiler (`http://sf.net/p/obfusc`) is for ANSI C [1] but the approach is generic. Our compiler's particular constructions are guided by the principle:

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

and, for that, a logic to reason about runtime entropy is required. The compiler's aim in ($\widetilde{\mathbb{h}}$) is to vary object codes on each recompilation of the same source code so that runtime traces vary to the maximal extent possible. This kind of 'chaotic' compilation is first described in [2]. The program semantics is not conserved, but the difference from the nominal semantics is controlled. That is, the semantics obtained differs by a planned (different) 'delta' at every register and memory location before and after every instruction. That is called an *obfuscation scheme*. It makes the code and the runtime trace harder to read, so 'obfuscation' is an appropriate term for it. The scheme is a secret initially known only to the compiler and in principle shared only with the user/owner of the code. Reasons why it may be desirable to obfuscate will not be gone into here as introducing the technology is the focus, but they include intellectual property protection and other goals [3]. How it works may be illustrated via the following simple loop:

$$\mathsf{while}\ \mathsf{x} < \mathsf{y} + \mathsf{z} + 1\ \mathsf{do}\ \{\mathsf{x} \leftarrow \mathsf{x} + 2;\ \mathsf{x} \leftarrow \mathsf{x} + 3;\ \}$$

Imagine new program variables X, Y, Z and shift them by different deltas from the program variables x, y, z at different points in the code as shown below:

$$\mathsf{while}\ \underbrace{\mathsf{X}+4}_{\mathsf{x}} < \underbrace{\mathsf{Y}+5}_{\mathsf{y}} + \underbrace{\mathsf{Z}+6}_{\mathsf{z}} + 1\ \mathsf{do}\ \{\underbrace{\mathsf{X}+7}_{\mathsf{x}} \leftarrow \underbrace{\mathsf{X}+4}_{\mathsf{x}} + 2;\ \underbrace{\mathsf{X}+4}_{\mathsf{x}} \leftarrow \underbrace{\mathsf{X}+7}_{\mathsf{x}} + 3;\ \}$$

The imagined relation $\mathsf{x} = \mathsf{X} + 4$ has to be the same at the end of the loop as at the beginning, but otherwise the choice is free. Simplifying:

$$\mathsf{while}\ \mathsf{X} < \mathsf{Y} + \mathsf{Z} + 8\ \mathsf{do}\ \{\mathsf{X} \leftarrow \mathsf{X} - 1;\ \mathsf{X} \leftarrow \mathsf{X} + 6;\ \}$$

(signed 2s complement comparison is translation-invariant). A user can see the first while loop execute and understand it as this while loop, working their imagination to make the substitutions. The lesson is that one program can be understood in many ways by an observer, who simply believes that the program variables are different by different deltas at different points in the program.

Conversely, a user intending the final while loop can instead obfuscate it to the first while loop, installing the given obfuscation scheme of deltas through the program, let it execute, and, discounting again the obfuscation, trust that it is the final while loop that has executed, which is really the one of interest.

The obfuscated codes (a) look approximately the same, having the same structure and differing only in program constants. Also (b) runtime traces 'look the same', with the same instructions in the same order reading and writing the same registers and memory locations (the end objective is to obfuscate at machine code level), while data varies from nominal by planned but arbitrary deltas, different at every point in the runtime trace and registers/memory, with the obvious provisos that:

deltas are equal across copy or skip, and where control paths meet. ($\underset{\sim}{\mathbb{h}}$)

That is necessary in order for computation to work properly. In particular, loops must have the same delta from nominal at either end, ready for a loop repeat.

This paper describes 'correct by construction' compilation for obfuscation as above following the principle ($\widetilde{\mathbb{h}}$). For the compiled programs, at any m points in the trace not related as in ($\underset{\sim}{\mathbb{h}}$), it is shown that variations with $32m$ bits of entropy occur, supposing a 32-bit machine. That quantifies the obfuscation.

The compiler's job is to vary the constants embedded in the machine code instructions so all feasible trace variations are exercised *equiprobably*. How it does that is summarised in Box 1: a new obfuscation scheme is generated at each recompilation. That is formally a set of vectors of *planned deltas from nominal values for the data per memory and register location, one such vector before and after each machine code instruction.* A declarative outline of the compile procedure is as follows: the compiler $\mathbb{C}[-]$ translates an expression e of type Expr that is to end up in register r at runtime to machine code mc of type MC and plans a 32-bit integer delta Δr (type Off) in r:

$$\begin{aligned} &\mathbb{C}[\text{-}]^r :: \text{Expr} \to (\text{MC}, \text{Off}) \\ &\mathbb{C}[e]^r = (mc, \Delta r) \end{aligned} \tag{1}$$

Let $s(r)$ be the value in register (or memory location) r in state s of the processor at runtime. The state is comprised by the values in registers and memory. Let $s(e)$ be the nominal (i.e., canonical, or at least standard, for some standard) evaluation of expression e. Running the code mc changes the state s to state s'

Box 1: An 'obfuscating' compiler does as follows:

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

that holds a value in r whose value differs by Δr from nominal. That is:

$$s \overset{mc}{\rightsquigarrow} s' \text{ where } s'(r) = s(e) + \Delta r \tag{2}$$

The compiler's choices of deltas Δr before and after each machine code instruction are known to the user/owner of the code, but not the processor or the operating system. Since the user knows the obfuscation scheme, they can create meaningful inputs for the compiled program and interpret the outputs.

Section 2 introduces a concrete, modified OpenRISC (`http://openrisc.io`) machine code instruction set as a target for obfuscating compilation as described above. It can alternatively be understood as a set of assembly-level macros to be implemented by the instructions of another platform. The small-step instruction semantics in Table 1 concretizes what is described broadly by $s \overset{mc}{\rightsquigarrow} s'$ in (2).

The obfuscating compiler technology illustrated in the discussion of this section is described less abstractly via the translation of three canonical source code constructions in 3.1 and 3.2. A pre-/post-condition Hoare program logic [4] for the calculus of deltas that the compiler must use to keep track of its code variations is introduced in 3.3, and in 3.4 it is modified to an obfuscation calculus for stochastic compiler-induced entropy. A spin-off is that one is able to reason about statements such as the covariance $\mathrm{cov}(\mathsf{x}, \mathsf{x}') = 0$ for a program variable x.

2 FxA Instructions

A 'fused anything and add' (FxA) [2] instruction set architecture (ISA) will be the specific compilation target here. The integer portion is shown in Table 1. The instruction set is adapted from OpenRISC ISA v1.1 (`http://openrisc.io/or1k.html`), which has about 200 instructions so FxA has about that many too. There are instructions for single and double precision integer operations, single and double floating point, and vector operations, all 32 bits long. Instructions access up to three 32 general purpose registers (GPRs), and one of those register operands may be replaced by a ('immediate') constant. A number of 32-bit 'prefixes' may precede a 32-bit instruction which has 16 bits of room for an onboard constant to provide room for 16 bits more of that constant and also any others that the instruction requires.

Table 1. FxA instruction set for encrypted work.

op.	*fields*	*mnem.*	*semantics*
add	$r_0\, r_1\, r_2\, k$	add	$r_0 \leftarrow r_1 + r_2 + k$
sub	$r_0\, r_1\, r_2\, k$	subtract	$r_0 \leftarrow r_1 - r_2 + k$
mul	$r_0\, r_1\, r_2\, k_0 k_1 k_2$	multiply	$r_0 \leftarrow (r_1 - k_1) * (r_2 - k_2) + k_0$
div	$r_0\, r_1\, r_2\, k_0 k_1 k_2$	divide	$r_0 \leftarrow (r_1 - k_1) \div (r_2 - k_2) + k_0$
...			
mov	$r_0\, r_1$	move	$r_0 \leftarrow r_1$
beq	$i\ r_1\, r_2\, k$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1 = r_2 + k$
bne	$i\ r_1\, r_2\, k$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1 \neq r_2 + k$
blt	$i\ r_1\, r_2\, k$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1 < r_2 + k$
bgt	$i\ r_1\, r_2\, k$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1 > r_2 + k$
ble	$i\ r_1\, r_2\, k$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1 \leq r_2 + k$
bge	$i\ r_1\, r_2\, k$	branch	if b then $pc \leftarrow pc+i,\ b \Leftrightarrow r_1 \geq r_2 + k$
...			
b	i	branch	$pc \leftarrow pc + i$
sw	$(k_0)r_0\ r_1$	store	$\mathrm{mem}[\![r_0 + k_0]\!] \leftarrow r_1$
lw	$r_0\ (k_1)r_1$	load	$r_0 \leftarrow \mathrm{mem}[\![r_1 + k_1]\!]$
jr	r	jump	$pc \leftarrow r$
jal	j	jump	$ra \leftarrow pc + 4;\quad pc \leftarrow j$
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.
i – prog. incr.; r – register content

3 Obfuscating Compilation

An obfuscating compiler as outlined in Section 1 works with a database $D : \mathrm{Loc} \to \mathrm{Off}$ containing (here 32-bit) integer delta offsets Δl (type Off) for data, indexed per register or memory location l (type Loc). That is varied by the compiler as it makes its pass through the source code. The delta Δl defines by how much the runtime data is to differ from nominal in l at that point in the program control graph, and database D is an *obfuscation scheme* at that point.

The database $L : \mathrm{Var} \to \mathrm{Loc}$ that maps source code variables to registers and memory will not be treated here but taken as given (and annotated as a superscript on the compiler symbol).

Taking databases D (type DB) and L into account, the expression compiler $\mathbb{C}[e]^r$ described in Section 1 that places the result value in target register r in fact will be written $\mathbb{C}^L[D : e]^r$ with type signature:

$$\mathbb{C}^L[\_ : \_]^r : \mathrm{DB} \times \mathrm{Expr} \to \mathrm{MC} \times \mathrm{Off} \tag{3}$$

where MC stands for machine code, a sequence of FxA instructions mc. The compiler aims to vary the deltas Δl of type Off equiprobably over the type across recompilations. The following paragraphs explain how it is feasible.

3.1 Expressions

To translate `x + y` where `x`, `y` are signed 32-bit integer source code variables, the compiler first emits machine code mc_1 as in (4a). At runtime that will put the

value of x in register $r_1 = L\mathsf{x}$ with offset delta $\Delta\mathsf{x}$ (a pair in DB×Expr is written $D : \mathsf{x}$ for readability here):

$$(mc_1, \Delta\mathsf{x}) = \mathbb{C}^L[D : \mathsf{x}]^{r_1} \tag{4a}$$

$$s_0 \overset{mc_1}{\rightsquigarrow} s_1 \ : \ s_1(r_1) = s_0(\mathsf{x}) + \Delta\mathsf{x} \tag{4b}$$

Small step semantics is from Table 1, with the value $s(r)$ in register r. To make that work formally, define the *nominal* value of a variable x in general as

$$s(x) \ =_{\text{DF}} \ s(Lx) - DLx \tag{5}$$

where $r = Lx$ is the location for the variable and Dr is the delta intended for that location at this point in the code. Then $\Delta r_1 = Dr_1$ and

$$s_1(\mathsf{x}) = s_0(\mathsf{x}) \tag{6}$$

has maintained the nominal value of variable x despite the numerical value in the register r_1 where it is located being different from that by $\Delta\mathsf{x}$.

The compiler next emits machine code mc_2. At runtime that will put the value of y in register $r_2 = L\mathsf{y}$ with offset delta $\Delta\mathsf{y}$:

$$(mc_2, \Delta\mathsf{y}) = \mathbb{C}^L[D : \mathsf{y}]^{r_2} \tag{7a}$$

$$s_1 \overset{mc_2}{\rightsquigarrow} s_2 \ : \ s_2(r_2) = s_1(\mathsf{y}) + \Delta\mathsf{y} \tag{7b}$$

As with x, that maintains the canonical value of y:

$$s_2(\mathsf{y}) = s_1(\mathsf{y}) \tag{8}$$

The compiler emits the FxA integer **add** instruction that at runtime adds the sum from r_1 and r_2 into r_0, plus an increment k:

$$\begin{aligned} \mathbb{C}^L[D : \mathsf{x} + \mathsf{y}]^{r_0} &= (mc_0, \Delta e) \\ mc_0 &= mc_1;\ mc_2;\ \textbf{add}\ r_0\ r_1\ r_2\ k \end{aligned} \tag{9a}$$

Setting the canonical value of a sum expression as $s(x + y) = s(x) + s(y)$, and choosing $k{=}\Delta e{-}\Delta\mathsf{x}{-}\Delta\mathsf{y}$, the expression gets the following value at runtime:

$$s_0 \overset{mc_0}{\rightsquigarrow} s_2 \ : \ s_2(r_0) = s_0(\mathsf{x}) + s_1(\mathsf{y}) + \Delta e \tag{9b}$$

The register r_0 in which x is located will not be touched by the code mc_2 that evaluates y, because a compiler has always to be careful about that, so $s_1(\mathsf{x}) = s_0(\mathsf{x})$ and the final value in r_0 is the nominal value of $\mathsf{x} + \mathsf{y}$ offset by delta Δe. The delta is independent of the two for x and y – the instruction constant k may be adjusted at will to suit.

3.2 Statements

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

$$\mathbb{C}^L[\_ : \_] : \mathrm{DB} \times \mathrm{Stat} \to \mathrm{DB} \times \mathrm{MC} \tag{10}$$

Consider an assignment $\mathsf{z}{=}\mathsf{x}+\mathsf{y}$ of the previous expression $\mathsf{x}+\mathsf{y}$ to a source code variable z, which the location database L binds in register $r_z{=}L\mathsf{z}$. Let $\mathsf{x}{+}\mathsf{y}$ be called e here. The compiler emits code mc_0 that evaluates expression e in register **t0** with (randomly chosen) offset Δe as described in (9a) with $\mathbf{t0} = r_0$. A short-form **add** instruction with semantics $r_z \leftarrow \mathbf{t0} + k$ is emitted:

$$\mathbb{C}^L[D_0 : \mathsf{z}{=}e] = D_1 : \ mc_0;\ \mathbf{add}\ r_z\ \mathbf{t0}\ k \tag{11a}$$

The compiler sets $k{=}\Delta r_{\mathsf{Z}}{-}\Delta e$ to choose delta $\Delta\mathsf{z}$ for z in $r_{\mathsf{Z}} = L\mathsf{z}$:

$$s_0 \overset{mc_0}{\rightsquigarrow} s_2 \overset{\mathrm{add}}{\rightsquigarrow} s_3 \ : \ s_3(r_z) = s_0(\mathsf{x}) + s_1(\mathsf{y}) + \Delta\mathsf{z} \tag{11b}$$

The database of offset deltas is updated from $D_0 r_{\mathsf{Z}}$ to $D_1 r_{\mathsf{Z}}{=}\Delta\mathsf{z}$ here, so that is

$$s_3(\mathsf{z}) = s_0(\mathsf{x}) + s_1(\mathsf{y}) \tag{12}$$

Again, the final delta $\Delta\mathsf{z}$ may be freely chosen independent of the others by choosing the instruction constant k as required.

3.3 Offset Calculus

A classical pre-/post-condition calculus [4] captures more handily the compiler's changes to the obfuscation scheme of deltas in each register and memory location as it makes its pass through the source code.

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

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

By the example (9b,11b), the $\Delta' r$, Δr are independently chosen as the compiler modifies (post-) vector Δ' to (pre-) vector Δ:

$$\{\Delta\}\ \mathsf{z} = e\ \{\Delta'\} \tag{13a}$$

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

The vectors Δ, Δ' are identical on the *complement* $\bar{\rho}$ of ρ.

Vectors. The Δ above are indexed by the entire type Loc of registers and memory locations but in practice only a small subset is needed for a program.

When pointers (memory addresses calculated dynamically) are involved we augment the type system of the source language so each is declared restricted to point into a particular global array as workspace:

$$\mathsf{int\ A[100];\ \ldots;\ \textbf{restrict}\ A\ int\ *\ ptr;}$$

That reduces the set of possible memory locations as indices of the Δ vectors (to the storage space for A in expressions involving $*$ptr here).

The predictable problem of not knowing at compile-time which entry in the array a pointer points to at run-time turns out not to be relevant: we are ultimately interested in the statistical variation of the content achieved by compiling stochastically, and it will ideally be uniformly the same for all array entries.

Conditionals. Source code conditionals are compiled to machine code branch instructions, but which branch is for true and which for false from boolean expression b can be deliberately varied by the compiler, which randomly chooses to generate either code for b or for $\neg b$ at each level of subexpression.

That compile procedure is detailed in [2], but it is not complicated. It has already been described: the 1/0 result b of each boolean subexpression is modified by a randomly chosen 1-bit delta δ to $b + \delta \mod 2$ as just related for arithmetic expressions with the difference that the delta for booleans is 1-bit, not 32-bit.

The compiler tracks the nominal value, but it is not determinable by an observer whether the intended semantics has the nominal **then** or **else** branch as the target of the branch instruction jump, just as it is not determinable which **while** loop was meant by the code author in the introductory example given in Section 1. The same technique is used in classic 'garbled circuits' [5] technology for obfuscating hardware logic circuit design, with the difference that circuits cannot be recursive and boolean expressions can be.

The upshot is that the appropriate pre-/post- logic is a classic nondeterministic choice. Let ρ be the registers written in e. The deduction is:

$$\frac{\{\Delta_1\}\ s_1\ \{\Delta'\} \quad \{\Delta_2\}\ s_2\ \{\Delta'\}}{\{\Delta\}\ \mathsf{if}\ \ (e)\ s_1\ \ \mathsf{else}\ \ s_2\ \{\Delta'\}} \tag{14a}$$

$$\Delta \supseteq \Delta_1|_{\bar{\rho}} \cup \Delta_2|_{\bar{\rho}} \tag{14b}$$

and Δ, Δ_1, Δ_2 are identical on $\bar{\rho}$, otherwise independent. The final offsets Δ' set by the compiler are equal at the end of both branches, as the code following it will be compiled supposing it receives the same scheme of offsets no matter which branch was executed.

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

are equally offset at loop start and end, i.e., $\Delta_1|_{\bar{\rho}}{=}\Delta_2|_{\bar{\rho}}{=}\Delta'|_{\bar{\rho}}$ in (14a),(14b):

$$\frac{\{\Delta\}\ s\ \{\Delta'\}}{\{\Delta\}\ \ \mathsf{do}\ \ s\ \mathsf{while}\ \ e\ \{\Delta'\}} \tag{15a}$$

$$\Delta \supseteq \Delta'|_{\bar{\rho}} \tag{15b}$$

By this, the compiler is free to set offsets $\Delta|_\rho$ and $\Delta'|\rho$ independently.

3.4 Obfuscation Calculus

Let f_r be the probability distribution of offset Δr from a nominal value v beneath the encryption in register r, as the compilation varies stochastically, so $\text{prob}(s(r) = v{+}d){=}\text{prob}(\Delta r{=}d){=}f_r(d)$, where s is the processor state.

Each Δr, $\Delta' r$ is a random variable with a probability distribution, giving the stochastic analogue below (16) of (13) . Let variable x be stored in register or memory location r_x, and similarly for y, z;

$$\begin{gathered}\{\Delta r_\mathsf{x} = \mathcal{X},\ \Delta r_\mathsf{y} = \mathcal{Y},\ \Delta r_\mathsf{z} = \mathcal{Z}\}\\ \mathsf{z} = \mathsf{x} + \mathsf{y}\\ \{\Delta' r_\mathsf{x} = \mathcal{X},\ \Delta' r_\mathsf{y} = \mathcal{Y},\ \Delta' r_\mathsf{z} = \mathcal{Z}'\}\end{gathered} \tag{16}$$

In particular, Δr_z and $\Delta' r_\mathsf{z}$ are independent random variables.

Let T be the runtime trace of a program. That is a sequence consisting of each instruction executed and the 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 $\mathbb{E}[-\log_2 f_T]$. The increase in entropy from T to T' (it cannot decrease with longer T) is informally the number of bits of unpredictable information added.

We will need only these two facts from information theory:

Proposition 1 *The flat distribution $f_x{=}1/k$ constant is the one with maximal entropy $\mathrm{H}(x){=}\log_2 k$, on a signal x with k values.*

Proposition 2 *Adding a maximal entropy signal to any random variable on a n-bit space (2^n values) gives another maximal entropy, i.e., flat, distribution.*

Proposition 1 identifies the maximal entropy as n on an n-bit space, achieved when each of the 2^n values is equally probable. That is a completely disordered, or chaotic, signal. Proposition 2 uses the known fact (Shannon) that the entropy of the sum of two signals is at least as great as that of either. The surprising inference is that the characteristics of any distribution on a finite point space are obliterated completely, not partially, by adding a 'chaotic' signal to it, i.e., one with flat, uniform distribution.

Below, the logic is worked through for this stochastic view of compilation for the three constructs already seen: assignment, conditionals, and **while** loops.

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

$$\{\Delta\}\ \mathsf{z} = e\ \{\Delta'\} \tag{17a}$$

but the Δ, Δ' are vectors of offsets Δr, $\Delta' r$ that are random variables as in (16).

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 condition is still $\Delta|_{\bar\rho}{=}\Delta'|_{\bar\rho}$ here. I.e.:

$$\Delta \supseteq \Delta'|_{\bar\rho} \tag{17b}$$

but there is more to come, because we suppose the compiler follows the principle ($\widetilde{\mathbb{h}}$) and that means each new random variable is independent with maximal entropy. Each represents the compiler's free choice of constant like k of (9a,11a) in 'an arithmetic instruction that writes' (**add** and **addi** respectively in those).

Let the trace entropy up to the assignment be $\mathrm{H}(T){=}h$. Writing to r_{z} with offset delta that is a new independent r.v. U increases the trace entropy to $\mathrm{H}(T'){=}h{+}\mathrm{H}(U)$. The offset is 32-bit, chosen with flat distribution by the compiler, as per ($\widetilde{\mathbb{h}}$), so $\mathrm{H}(U){=}32$. There are $n{+}1$ registers $r_0, \ldots, r_n$ that are written independently, including the one holding target variable z, so entropy increases by $32(n{+}1)$ bits:

$$\{\mathrm{H}(T) = h\}\ \ \mathsf{z} = e\ \ \{\mathrm{H}(T') = h + 32(n{+}1)\} \tag{17c}$$

In more general form, writing $\Phi[A/B]$ for substitution of A for B in predicate Φ:

$$\{\Phi[\mathrm{H}(T) + 32(n+1)/\mathrm{H}(T)]\}\ \ \mathsf{z} = e\ \ \{\Phi'[\mathrm{H}(T')]\}$$

Put $\mathrm{H}(T) = h + 32(n+1)$ for Φ to obtain (17c) again.

That is the common case. But where the instruction that writes has already appeared once earlier in the trace, the offset delta it introduces is already known, and the increment in trace entropy is zero this time:

$$\{\mathrm{H}(T) = h\}\ \ \mathsf{z} = e\ \ \{\mathrm{H}(T') = h\} \tag{17c0}$$

Alternatively:

$$\{\Phi[\mathrm{H}(T)]\}\ \ \mathsf{z} = e\ \ \{\Phi'[\mathrm{H}(T')]\}$$

Remark 1. Whatever the expression e, provided it contains some arithmetic (even $\mathsf{z}+0$ will do), z' is independent of z is implied, as stated in Section 1.

Here 'z' is understood to refer to the exact value in the register r_{z} where z is located. By (17c), If $H(T) = h$ before the assignment, then it is $H(T') = h + 32(n+1)$ after the assignment, where $n+1$ in the number of 'arithmetic instructions that write' that appear in the machine code, including one that last writes to r_{z}. That can only happen if each such instruction introduces 32 bits of entropy to the trace because that is maximal per instruction, and therefore the last write to r_{z} introduces 32 bits of entropy. That means it adds an independent

uniformly distributed 32-bit delta δ. Then $\mathrm{prob}(r_{\mathsf{z}} = i \wedge r'_{\mathsf{z}} = j) = \mathrm{prob}(r_{\mathsf{z}} = i \wedge r'_{\mathsf{z}} - r_{\mathsf{z}} = j - i) = \mathrm{prob}(r_{\mathsf{z}} = i)\mathrm{prob}(r'_{\mathsf{z}} - r_{\mathsf{z}} = j - i | r_{\mathsf{z}} = i) = \mathrm{prob}(r_{\mathsf{z}} = i)\mathrm{prob}(\delta = j - i | r_{\mathsf{z}} = i) = \mathrm{prob}(r_{\mathsf{z}} = i)/2^{32} = \mathrm{prob}(r_{\mathsf{z}} = i)\mathrm{prob}(r'_{\mathsf{z}} = j)$ since r'_{z} is distributed uniformly too (as the sum with a maximal entropy difference signal δ). So r_{z} and r'_{z} are independent. and their covariance is 0, i.e., $\mathrm{cov}(\mathsf{z}, \mathsf{z}') = 0$.

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

$$\frac{\{\Delta_1\}\ s_1\ \{\Delta'\} \quad \{\Delta_2\}\ s_2\ \{\Delta'\}}{\{\Delta\}\ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \{\Delta'\}} \tag{18a}$$

$$\Delta \supseteq \Delta_1|_{\bar{\rho}} \cup \Delta_2|_{\bar{\rho}} \tag{18b}$$

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 b, say $32n$ bits of entropy from n writes to n registers, plus the entropy from the trace through a branch:

$$\frac{\{\mathrm{H}(T'){=}h{+}32n\}\ s_1\ \{\Theta\} \quad \{\mathrm{H}(T'){=}h{+}32n\}\ s_2\ \{\Theta\}}{\{\mathrm{H}(T) = h\}\ \ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \ \{\Theta\}} \tag{18c}$$

or, in more generic form:

$$\frac{\{\Phi'[\mathrm{H}(T')]\}\ s_1\ \{\Theta\} \quad \{\Phi'[\mathrm{H}(T')]\}\ s_2\ \{\Theta\}}{\{\Phi[\mathrm{H}(T) + 32n/\mathrm{H}(T)]\}\ \ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \ \{\Theta\}}$$

Put $\mathrm{H}(T) = h + 32n$ for Φ in the above to get (18c).

To make that deduction valid, the compiler must even up the number of 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 (see below).

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

$$\{\mathrm{H}(T) = h\}\ \ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \ \{\mathrm{H}(T') = h\} \tag{18c0}$$

If it branches a different way from the first time, 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 offsets are constrained in (14b) to agree with the offsets in the other branch, which are already known. So those m do not count:

$$\frac{\{\mathrm{H}(T){=}h\}\ s_1\ \{\Theta\}\ \{\mathrm{H}(T){=}h\}\ s_2\ \{\Theta\}}{\{\mathrm{H}(T){+}32m = h\}\ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \{\Theta\}} \tag{18c1}$$

Those final m instructions that synchronise the offsets with the other branch get a special name:

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.*

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

$$\frac{\{\Delta\}\ s\ \{\Delta'\}}{\{\Delta\}\ \ \mathsf{do}\ \ s\ \mathsf{while}\ \ (b)\ \{\Delta'\}} \tag{19a}$$

$$\Delta \supseteq \Delta'|_{\bar{\rho}} \tag{19b}$$

That means $\Delta r{=}\Delta' r$ for $r{\notin}\rho$. Those distributions are equal because the values are equal for $r{\notin}\rho$, by (17b).

A trace over the loop is always the same length between recompilation and recompilation, 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 b 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:

$$\frac{\{\mathrm{H}(T){+}32m = h\}\ s\ \{\mathrm{H}(T') = h\}}{\{\mathrm{H}(T){+}32(n{+}m){=}h\}\ \ \mathsf{do}\ \ s\,\mathsf{while}\ \ b\ \{\mathrm{H}(T'){=}h\}} \tag{19c}$$

Put $\mathrm{H}(T) + 32m = h$ for Φ in the following generalisation to get (19c):

$$\frac{\{\Phi'[\mathrm{H}(T')]\}\ s\ \{\Theta\}}{\{\Phi[\mathrm{H}(T){+}32n/\mathrm{H}(T)]\}\ \ \mathsf{do}\ \ s\,\mathsf{while}\ \ b\ \{\Theta\}}$$

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

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

$$\{\mathrm{H}(T) = h\}\quad \mathsf{do}\ \ s\ \mathsf{while}\ \ b\ \ \{\mathrm{H}(T) = h\} \tag{19c0}$$

The (red) equations are an obfuscation calculus for trace entropy when compilation follows the principle ($\widetilde{\mathbb{h}}$). In summary, counting up via the rules above:

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' are identified with those instructions that read first time a location that has not yet been written in the trace.

A successfully obfuscating compiler's constructions must recruit every emitted arithmetic instruction that writes to the task of freely varying the offsets in data beneath the encryption in register and memory locations. Otherwise it is not doing as much as it could to contribute to variability in the trace. The sole restriction on the compiler 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 would go wrong otherwise. The following characterises the best way of compiling to produce this kind of obfuscation:

Proposition 3 *The entropy in the run-time traces induced by a compiler following the principle* ($\widetilde{\mathbb{h}}$) *is maximal among compositional strategies for varying the constant parameters in the compiled machine code while it still works correctly.*

The reasoning is that if the compiler works compositionally then it does not know the context in which its constructions are used, so it must suppose that data that is written will later be read, and so must arrange for synchronisation between all final offset deltas in different branches of conditionals, even for locations that in fact are never read afterwards. If the data is never read, synchronisation could be done without and total entropy would be increased when both branches are traversed in the trace. But for working compositionally, a compiler could put more entropy into the trace, but it would be for data that serves no purpose because it is never read.

The only other way to put more entropy in is to vary individual instructions more, but that is impossible for a compiler that already implements ($\widetilde{\mathbb{h}}$).

The proposition implies that on a 32-bit platform a full 32 bits of entropy per datum are provided by the compiler that follows ($\widetilde{\mathbb{h}}$):

Corollary 1 *The probability across different compilations that any particular 32-bit value x is in a given register or memory location at any given point in the trace at runtime is uniformly* $1/2^{32}$.

But what of two or more data values at different points in the trace? That depends on how they are connected computationally. If they are input and output of a copy instruction, they will be correlated.

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

If data is taken at two (m) independent points, the variation is maximal:

Theorem 1 *The probability across different compilations that any m particular 32-bit values have values x_i in given register or memory locations 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. The theorem asserts that with ($\widetilde{\mathbb{h}}$) as a guide, as much trace data as one cares to observe is maximally unpredictable across recompilations, as far as is computationally possible.

Remark 2. The theorem speaks to an argument about obfuscation in plain sight. It is impossible for any observer to know what the variables X, Y, Z are in the **while** loop given in Section 1, because they exist only in the mind of the code author. The problem is that the observer has only to duplicate the form of the

loop and run it to get a sequence of values x_0, x_1, x_2, etc. in x and y and z that are duplicated at some unknown offset as X_0, X_1, X_2, etc. by X, Y, Z. Thus $X_1 - X_0$, $X_2 - X_1$, etc. are known to the observer, being equal to respectively $x_1 - x_0$, $x_2 - x_1$, etc. The theorem says that if the code author is to avoid the observer being able to know even that much, then code with loops is out.

Suppose an observer claims to have a polynomial time (in the number of bits n in a word on the platform, hitherto supposed to be 32) method of working out what the code author means to have in their variable X at some identified point in the trace of program P. The point of interest to the observer may even move (polynomially) with n, being specifed, say as 'the point of last change in the first n^3 steps'. The observer knows program P and may even have suggested it. The observer also can see the compiled maximal entropy code $\mathbb{C}[P]$, and see it running and probe it deeply by running it themself.

The code author then readies a sequence of maximal entropy compilations $\mathbb{C}[P_n]$ of P, with the nth being for a n-bit platform as target, and with P having been partially or completely unrolled as P_n with no loops in at least the first e^n (i.e., super-polynomially many) machine code instructions. If the program predictably ends before then, just unroll completely. The observer is invited to apply their method and predict what is meant by the values at their chosen points in the trace(s) of these programs, which differ only in consequence of the number n of bits in a word on the platform .

The theorem implies the observer's method cannot exist. There are 'no loops' (i.e., no dependencies, per the wording of the theorem) in the part of the trace the observer has time to examine. The theorem says the compiler will have arbitrarily and independently varied what is meant by the values throughout that length of the trace by varying the deltas from the nominal value independently across each instruction in turn. The observer, seeing a 1, cannot tell if 2 was meant.

The credibility of the argument is supported by the trivial case in which the program unrolls completely. Then it is equivalent to a logic circuit in hardware. It is known from the theory of Yao's garbled circuits [5] that the intended values cannot be deciphered without the garbling scheme, which equates to an obfuscation scheme of deltas here via '+1 mod 2' being boolean negation on a 1-bit logic value, while '+0 mod 2' leaves the logic value unchanged.

4 Implementation

Our own prototype compiler `http://anonymised.url` following 3.4 is for ANSI C [1], 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:`

Table 2. Trace for Ackermann(3,1), result 13.

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

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

`//jhauser.us/arithmetic/TestFloat.html`. A trace[3] of the Ackermann function[4] [6] is shown in Table 2.

5 Conclusion

A formal obfuscation calculus for programs is set out that quantifies the entropy in the data beneath the encryption in a runtime trace, where compilation is stochastic (and execution is not). The way to maximise the entropy is to follow the principle for compiler constructions that every arithmetic instruction that writes should be varied maximally across recompilations.

References

1. ISO/IEC, "Programming languages – C," Int. Org. for Standardization, 9899:201x Tech. Rep. n1570, Aug. 2011, JTC 1, SC 22, WG 14.
2. 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.
3. B. Barak, "Hopes, fears, and software obfuscation," *Commun. ACM*, vol. 59, no. 3, pp. 88–96, Feb. 2016.
4. C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–80, 1969.
5. A. C.-C. Yao, "How to generate and exchange secrets," in *27th Ann. Symp. Found. Comp. Sci.* IEEE, 1986, pp. 162–167.
6. Y. Sundblad, "The Ackermann function: a theoretical, computational, and formula manipulative study," *BIT Num. Math.*, vol. 11, no. 1, pp. 107–19, 1971.

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

[4] 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)); }.