EasyChair Preprint

№ 1612

A Calculus of Chaos in Stochastic Compilation: Engineering in the Cause of Mathematics

Peter T. Breuer and Simon Pickin

October 9, 2019

A Calculus of Chaos in Stochastic Compilation

Engineering in the Cause of Mathematics

Peter T. Breuer[1,2] and Simon J. Pickin[3]

[1] Hecusys LLC, GA, USA ptb@hecusys.com
[2] London South Bank University, UK
[3] Universidad Complutense de Madrid, Spain simon.pickin@fdi.ucm.es

Abstract. An unexpected outcome from an open project to develop a 'chaotic' compiler for ANSI C is described here: a trace information entropy calculus for stochastically compiled programs. A stochastic compiler produces randomly different object codes every time it is applied to the same source code. This calculus quantifies the entropy introduced into run-time program traces by a compiler that aims for the maximal possible amount, furnishing a definition and proof of security for encrypted computing (Turing-complete computation in which data remains in encrypted form throughout), where the status was unknown.

Keywords: Computer Security · Encrypted computing · Program logic.

1 Introduction

This article describes a program calculus that quantifies the entropy introduced into a run-time trace by a stochastic compiler, developed as part of an open source project (`http://sf.net/p/obfusc`) to develop a secure computing infrastructure. To be clear from the outset, the stochastic element here occurs not in the execution of a program but in its compilation. The software part of the project aims to develop a complete tool-chain (compiler, assembler, linker, loader) for *encrypted computing* [1] platforms [2,3,6,9,16,21]. Those are general purpose processors in which data remains in encrypted form throughout processing. Obviously computing encrypted is not less 'secure' than no encryption at all, but how secure has had little formal backing. For example, [5] exhibits a program for an encrypted computing platform that perfectly decrypts ciphertexts back to plaintext and, intuitively, the constants in it 'ought' to be as hard to guess as the encryption is to crack, but that needs proof.

Also introduced in [5] are properties of the machine code instruction set architecture (ISA) necessary for a secure runtime environment. Important is *malleability*: the constants in each machine code instruction may be varied to offset independently by any amount its inputs and outputs. The data beneath the encryption in a program trace can then be varied arbitrarily while the program black-box semantics remains the same, the code remains the same apart from the varied constants, and the same sequence of instructions appears in the trace [4]. Intuitively that means an attacker could never be sure they have

'cracked the encryption', as their solution is one of many.

This paper will only touch on security and focuses instead on a pure computer science problem: how to get the maximum possible variation into run-time traces via stochastic compilation, and state that with assurance. Introducing maximum variation will be called ***chaotic compilation***. It was not known if it is possible. This paper quantifies the notion and shows how to guarantee a compiler gets it right via a formal logic. That is then used to answer the mathematical question 'is secure computing possible', once it has been formalised appropriately. It is a case of engineering in the cause of mathematics, rather than vice versa.

The existing security concept applicable here is classic 'Holy Grail' *cryptographic semantic security* [12] for encryptions, best known via the game theory version of it [13]: there is no method of attack M of polynomial time complexity in the encryption block size n that infinitely often does non-negligibly better than chance (i.e., probability $1/2 + B$ for $B > 0$) at guessing the value of a given bit of data beneath the encryption, as $n \to \infty$. In the encrypted computing context, the encryption block size n is the same as the hardware word size n, the size of the processor's registers. The plaintext word beneath the encryption is constant, typically 32 bits, but the hardware word may be 128, 256 bits or more. IBM's work on (non-Turing complete) computation over fully homomorphic encryptions (FHEs; encryptions $\mathcal{E}$ that respect addition and multiplication, with $\mathcal{E}[a+b] = \mathcal{E}[a]+\mathcal{E}[b]$ and $\mathcal{E}[a*b] = \mathcal{E}[a]*\mathcal{E}[b]$) [8,10] uses custom vector machines with word sizes in the millions of bits and atomic operations that take on the order of one second [11]. But the word size n cannot vary arbitrarily, because hardware cannot change, so the concept must be tested mathematically.

The similar concept this project has created for encrypted computing is *relative semantic security*: there is no successful method of attack M that (i) has polynomial-time complexity in the number of bits n in the hardware word on the encrypted computing platform, and (ii) reveals the run-time data beneath the encryption, given (iii) there is no such method that is successful against the encryption. 'Success' means the method has probability of guessing right on each bit it reports that beats chance by a margin that does not tend to 0 as $n \to \infty$. Formally, if T_n is the trace of the program on an encrypted computing platform with an n-bit word and b is the targeted bit beneath the encryption, and (i) the worst case running time of M is $\mathbf{O}n^k$ for some k, then (ii) $\text{prob}[M(T_n) = b] > 1/2 + B$ for some $B > 0$ infinitely often as $n \to \infty$ is impossible, *provided* (iii) there is no such method M' that works against the encryption alone (classic cryptographic semantic security). Read $1/2^m+B$ in (ii) for guessing $m>1$ bits at a time.

The argument for that is sketched in more detail in Section 6. It follows from a quantification of the entropy induced in a program trace by a chaotic compiler. The formal method is a program logic expressing the trace entropy. Extant program logics with stochastic aspects [14,18] all, as far the authors know, deal with run-time randomness and we have not been able to discover any that deal with compile-time randomness. Program logics are abstractions of program semantics. In this case the aspect of interest is how much the program trace varies when the same program is compiled to make it vary 'as much as possible' from

recompilation to recompilation. If the compiler is successful, the variation it introduces statistically swamps other information content in the trace, meaning that any attack is effectively against statistically independent random data, which reduces it to an attack against the encryption alone.

This paper is organised as follows. Section 2 gives an informal introduction to the trick of program semantics and the maximum ($\widetilde{\mathbb{h}}$) and minimum ($\underset{\sim}{\mathbb{h}}$) entropy compilation principles. Section 3 gives a more formal overview of stochastic compilation and introduces a modified OpenRISC (`http://openrisc.io`) machine code instruction set as target for chaotically stochastic compilation. Compilation is described in Section 4 in 4.1 and 4.2. A pre-/post-condition Hoare program logic [15] for the calculus of differences that a compiler must use to track its own code variations is introduced in 4.3, and in 4.4 it is modified to an information entropy ('chaos') calculus for run-time traces. Section 5 gives examples and Section 6 sketches the security argument.

2 Overview of the Technical Foundation

The source language is ANSI C [17] here but the approach is generic. Chaotic compilation is guided by the principle:

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

The mechanism for that (introduced in [4]) is as follows: the object-code is to differ from the nominal semantics beneath the encryption by a planned and different random 'delta' at every register and memory location before and after every instruction. That is a *difference scheme*.

Definition 1 *A* difference scheme *is a set of vectors of* deltas from nominal for the data per memory and register location, one vector per point in the control graph of the machine code*, i.e., before and after each machine code instruction.*

The scheme is generated by the compiler and is shared with the user/owner of the code, so they may validate or verify the run-time trace and create inputs for the running program and read outputs from it. The restriction to arithmetic instructions in ($\widetilde{\mathbb{h}}$) is because copy and goto must work normally.

A thought experiment illustrates the trick of the mechanism. Consider the following pseudo-code 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, shifted 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 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 of 4, 5, 6, 7 is free. Simplifying, that is:[4]

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

An observer can watch the first while loop execute and understand it as the second loop. Conversely, a user intending the second while loop can instead execute the first, with the translations above in mind.

A stochastic compiler systematically does the above, but at the object code level. It introduces different deltas like 4, 5, 6, 7 above at every register and every memory location, per machine code instruction. A summary is that the object codes generated from the same source code:

(a) all have the same structure, differing only in the constant parameters embedded in the individual machine code instructions; also
(b) run-time traces have the same instructions (modulo (a)) in the same order reading and writing the same registers and memory locations; but
(c) data varies from nominal by planned but randomly chosen and arbitrary deltas, different at every point in the run-time trace and registers/memory.

The catch is that, as for the $\mathsf{x} = \mathsf{X} + 4$ in the thought experiment:

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

That is necessary in order for computation to work properly.

Compilation must be systematic, or it will produce neither the intended semantics nor properties. So this paper first (Section 4) describes 'correct by construction' compilation along the lines of the thought experiment above, introducing the 'deltas' of a difference scheme as compile-time parameters.

The question is if those can be chosen without restriction as ($\widetilde{\mathbb{h}}$) demands. The induced variation in the run-time program traces is measurable as (information-theoretic) entropy.[5] What ($\widetilde{\mathbb{h}}$) expresses is that at every instruction in the run-time trace where, say, a 32-bit value is written beneath the encryption, the chaotic compiler should introduce a delta with 32 bits of entropy in the choice available. Shannon's theorem [19] holds that adding one signal to another in fixed length arithmetic does not decrease entropy. Here, one input is the compiler's, the other is the programmer's. When the compiler's has maximal entropy (32 bits) then the combined signal also has maximal entropy and the information from the programmer has been statistically swamped and an observer cannot infer any deterministic relation or statistical tendency from the programmer's input. The two inputs are (i) not separately visible to a run-time observer, having been combined at compile time (and in encrypted computing the unencrypted form of the data is itself not visible), and (ii) the programmed data can be recovered afterwards by the intended user with the help of the difference scheme.

But ($\underset{\sim}{\mathbb{h}}$) acts to constrain entropy. Section 4 will show that, in a chaotically compiled program, at any m points in the trace *not* related as in ($\underset{\sim}{\mathbb{h}}$), variations

[4] Signed 2s complement comparison is translation-invariant. I.e., $x<y$ iff $x+k<y+k$.
[5] Entropy is formally the stochastic expectation $H = -E[\log_2 p_i]$ of the probability p_i of the possible observations i, thus $H = -\sum_i p_i \log_2 p_i$ with $0 \leq p_i \leq 1$ and $H \geq 0$.

Box 1: A stochastic (expression) compiler for encrypted computing.

The compiler $\mathbb{C}[-]$ translates an expression e of type Expr that should end up in register r at run-time to machine code mc of type MC and plans a 32-bit integer delta Δr (type Off) for it 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 run-time. The state is comprised by the values in registers and memory. Let $\ulcorner e \urcorner s$ be a nominal evaluation of expression e in state s.[a] Running the code mc changes the state s after several steps to state s' that holds a value in r whose value differs by Δr from the nominal value of the expression. That is:

$$s \overset{mc}{\rightsquigarrow} s' \text{ where } s'_r = \mathcal{E}[\mathcal{D}[\ulcorner e \urcorner s] + \Delta r] \tag{2}$$

where $\mathcal{E}$ is encryption (it may be trivial), $\mathcal{D}$ is decryption.

[a] If source code variable x is in register r with delta Δr, then the nominal value $\ulcorner \mathsf{x} \urcorner s = \mathcal{E}[\mathcal{D}[s_r] - \Delta r]$, $\ulcorner e_1 + e_2 \urcorner s = \mathcal{E}[\mathcal{D}[\ulcorner e_1 \urcorner s] + \mathcal{D}[\ulcorner e_2 \urcorner s]]$, etc.

with $32m$ bits of entropy are produced among the traces, on a 32-bit machine. For each pair of points related as in ($\mathbb{h}$), entropy reduces by 32 bits. That analysis leads to the argument (Section 6) that there is no successful[6] method of attack on the run-time data with polynomial-time complexity in the number of bits n in a processor word on an encrypted computing platform, given there is no successful method against the encryption in use (which has an n-bit block-size).

3 Overview of Stochastic Compilation

The action of a stochastic compiler (Box 2) parallels (a-c) of Section 1: (A) the constants embedded in the machine code instructions are varied so (B) all feasible trace variations are exercised (C) *equiprobably*, because an equiprobable distribution over the full range of values (uniquely) has maximal entropy.

The mechanism is to generate a new difference scheme at each recompilation. The principle is described in Box 1, where compilation of pure expressions is summarised. The Δr is the entry in the difference scheme for register r at that point in the program. The difference scheme is known to the code user alone

A reduced instruction set (RISC) 'fused anything and add' (FxA) [4] -style ISA will be the specific compilation target here, originally adapted from OpenRISC v1.1 (`http://openrisc.io/or1k.html`), The part needed for the purposes

[6] 'Success' is stochastic: the method has probability of being right on each bit that beats chance by a ('non-negligible') margin B that does not tend to 0 as $n \to \infty$.

Box 2: A stochastically 'chaotic' compiler implements the following strategy:

(A) *change only program constants, generating an arrangement of planned* deltas from nominal *for instruction inputs and outputs* (a difference scheme);
(B) *leave run-time traces unchanged, apart from differences in the program constants* (A) and *run-time data*;
(C) *equiprobably generate all possible difference schemes satisfying* (A), (B).

Table 1. RISC 'FxA' machine code instruction set.

op.	*fields*	*mnem.*	*semantics*
add	$\mathsf{r}_0\,\mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	add	$r_0 \leftarrow \mathcal{E}[\mathcal{D}r_1 + \mathcal{D}r_2 + k]$
sub	$\mathsf{r}_0\,\mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	subtract	$r_0 \leftarrow \mathcal{E}[\mathcal{D}r_1 - \mathcal{D}r_2 + k]$
mul	$\mathsf{r}_0\,\mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k_0\mathcal{E}k_1\mathcal{E}k_2$	multiply	$r_0 \leftarrow \mathcal{E}[(\mathcal{D}r_1 - k_1) * (\mathcal{D}r_2 - k_2) + k_0]$
div	$\mathsf{r}_0\,\mathsf{r}_1\,\mathsf{r}_2\,k_0k_1k_2$	divide	$r_0 \leftarrow \mathcal{E}[(\mathcal{D}r_1 - k_1) \div (\mathcal{D}r_2 - k_2) + k_0]$
...			
mov	$\mathsf{r}_0\,\mathsf{r}_1$	move	$r_0 \leftarrow r_1$
beq	$i\ \mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	branch	if b then $pc \leftarrow pc+i$, $b \Leftrightarrow \mathcal{D}r_1 = \mathcal{D}r_2 + k$
bne	$i\ \mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	branch	if b then $pc \leftarrow pc+i$, $b \Leftrightarrow \mathcal{D}r_1 \neq \mathcal{D}r_2 + k$
blt	$i\ \mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	branch	if b then $pc \leftarrow pc+i$, $b \Leftrightarrow \mathcal{D}r_1 < \mathcal{D}r_2 + k$
bgt	$i\ \mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	branch	if b then $pc \leftarrow pc+i$, $b \Leftrightarrow \mathcal{D}r_1 > \mathcal{D}r_2 + k$
ble	$i\ \mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	branch	if b then $pc \leftarrow pc+i$, $b \Leftrightarrow \mathcal{D}r_1 \leq \mathcal{D}r_2 + k$
bge	$i\ \mathsf{r}_1\,\mathsf{r}_2\,\mathcal{E}k$	branch	if b then $pc \leftarrow pc+i$, $b \Leftrightarrow \mathcal{D}r_1 \geq \mathcal{D}r_2 + k$
...			
b	i	branch	$pc \leftarrow pc + i$
sw	$(\mathcal{E}k_0)\mathsf{r}_0\ \mathsf{r}_1$	store	$\mathrm{mem}[\![\mathcal{E}[\mathcal{D}r_0 + k_0]]\!] \leftarrow r_1$
lw	$\mathsf{r}_0\ (\mathcal{E}k_1)\mathsf{r}_1$	load	$r_0 \leftarrow \mathrm{mem}[\![\mathcal{E}[\mathcal{D}r_1 + k_1]]\!]$
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.
i – prog. incr.
r – register content
$\mathcal{E}$ – encryption
$\mathcal{D}$ – decryption

of this paper is exhibited in Table 1. The general pattern of the ISA is that instructions access up to three 32 general purpose registers (GPRs), and one of those may instead be an ('immediate') constant embedded in the instruction.

The important feature is: every arithmetic instruction has constants that the compiler may vary to displace each instruction's inputs and outputs independently ('malleability'). Addition and branch instructions would suffice for Turing completeness (c.f. the Fractran language [7] which has just those primitives).

4 Chaotic Compilation

The compiler works with *difference scheme sections* $D : \mathrm{Loc} \to \mathrm{Off}$ with integer entries Δl (type Off), indexed per register or memory location l (type Loc). A complete difference scheme $\{D_p \mid p \in P\}$ has one of those per point p in the (object code) program P's control graph, i.e., before and after every machine code instruction. The delta Δl is how much the run-time data is to differ from nominal in l at point p.

A database $L : \mathrm{Var} \to \mathrm{Loc}$ that maps source code variables to registers and memory will be assumed. Then the expression compiler $\mathbb{C}[e]^r$ described in Box 1 that puts a result in register r is more exactly $\mathbb{C}^L[D : e]^r$ of type:

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

where Dsect is the type of D, MC is the type of machine code, a sequence of (FxA) instructions mc. The compiler aims to vary the deltas Δl equiprobably over the full range across recompilations. The following paragraphs explain how.

4.1 Pure Expressions

How source code is translated has to be shown in some detail in order to confirm or deny ($\widetilde{\mathbb{h}}$), because every time an 'instruction that writes' is emitted, it must be checked if it can be varied by the compiler to the maximum extent possible. Compilers work compositionally, so structural induction suffices for that. For pure expressions, every operation in it requires that the operands be in registers and a single machine code instruction then acts on them arithmetically and writes the result into another register. That instruction must be varied.

Translating x+y where x, y are signed 32-bit integer source code variables, the compiler emits machine code mc_1 as in (4a) that at run-time puts the value of x in register $r_1{=}L\mathsf{x}$ with offset delta Δr_1 (a pair $(D, \_)$ is written $D : \_$ here):

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

By inductive hypothesis, that is the nominal value plus the target register' delta:

$$s_0 \overset{mc_1}{\rightsquigarrow} s_1 \;:\; s_1\, r_1 = \mathcal{E}[\mathcal{D}[\ulcorner \mathsf{x} \urcorner\, s_0] + \Delta r_1] \tag{4b}$$

The small step semantics is from Table 1, with s_r or $s\,r$ for the value in register r in state s of the processor. The nominal value of the variable, defined in footnote a of Box 1, is preserved as the state changes from s_0 to s_1 through mc_1:

$$\ulcorner \mathsf{x} \urcorner\, s_1 = \mathcal{E}[\mathcal{D}[s_0\, L\mathsf{x}] - D\, L\mathsf{x}] = \mathcal{E}[\mathcal{D}[s_0\, r_1] - \Delta r_1] = \ulcorner \mathsf{x} \urcorner\, s_0 \tag{4c}$$

By induction too, machine code mc_2 for y is emitted preserving its nominal value:

$$(mc_2, \Delta\mathsf{y}) = \mathbb{C}^L[D : \mathsf{y}]^{r_2} \tag{5a}$$

$$s_1 \overset{mc_2}{\rightsquigarrow} s_2 \;:\; s_2\, r_2 = \mathcal{E}[\mathcal{D}[\ulcorner \mathsf{y} \urcorner\, s_1] + \Delta r_2] \tag{5b}$$

$$\ulcorner \mathsf{y} \urcorner\, s_2 = \ldots = \ulcorner \mathsf{y} \urcorner\, s_1 \tag{5c}$$

The compiler then emits the extra **add** instruction that at run-time sums r_1 and r_2 into r_0 with an increment k, a constant embedded in the instruction:

$$\begin{aligned} \mathbb{C}^L[D : \mathsf{x} + \mathsf{y}]^{r_0} &= (mc_0, \Delta e) \\ mc_0 &= \ulcorner mc_1;\; mc_2;\; \mathbf{add}\; r_0\; r_1\; r_2\; k \urcorner \end{aligned} \tag{6a}$$

Choosing $k{=}\Delta r_0{-}\Delta r_1{-}\Delta r_2$, the following value goes in r_0 at run-time:

$$s_0 \overset{mc_0}{\rightsquigarrow} s_2 \ : \ s_2\, r_0 = \mathcal{E}[\mathcal{D}[\ulcorner \mathsf{x} \urcorner\, s_0] + \mathcal{D}[\ulcorner \mathsf{y} \urcorner\, s_1] + \Delta r_0] \tag{6b}$$

$$= \mathcal{E}[\mathcal{D}[\ulcorner \mathsf{x} + \mathsf{y} \urcorner\, s_2] + \Delta r_0] \tag{6c}$$

The nominal value plus a delta ends up in register r_0 and the delta Δr_0 is independently and arbitrarily chosen by the compiler via its choice of k. The induction shows ($\widetilde{\mathbb{h}}$) is satisfied for pure expressions.

4.2 Statements

Let Stat be the type of statements, then compiling a statement changes the deltas and produces a new difference scheme section, as well as machine code:

$$\mathbb{C}^L[\_ : \_] : \text{Dsect} \times \text{Stat} \to \text{Dsect} \times \text{MC} \tag{7}$$

Consider an assignment $\mathsf{z}{=}\mathsf{x} + \mathsf{y}$ of 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 Δr_0 as described in (6a) with $\mathbf{t0} = r_0$. A short-form **add** instruction with semantics $r_z \leftarrow \mathbf{t0} + k$ is emitted to move that to r_z:

$$\mathbb{C}^L[D_0 : \mathsf{z}{=}e] = D_1 : \ulcorner mc_0;\ \mathbf{add}\ r_z\ \mathbf{t0}\ k \urcorner \tag{8a}$$

The compiler sets $k{=}\Delta r_\mathsf{z}{-}\Delta r_0$ to choose delta Δr_z arbitrarily:

$$s_0 \overset{mc_0}{\rightsquigarrow} s_2 \overset{\text{add}}{\rightsquigarrow} s_3 \ : \ s_3\, r_z = \mathcal{E}[\mathcal{D}[\ulcorner \mathsf{x} + \mathsf{y} \urcorner\, s_2] + \Delta r_\mathsf{z}] \tag{8b}$$

The difference scheme section is updated at r_z from $D_0 r_\mathsf{z}$ to $D_1 r_\mathsf{z}{=}\Delta r_\mathsf{z}$, so:

$$\ulcorner \mathsf{z} \urcorner\, s_3 = \ulcorner \mathsf{x} + \mathsf{y} \urcorner\, s_2 \tag{8c}$$

The final delta $\Delta r_\mathsf{z} = D_1 L\mathsf{z}$ may be freely and independently chosen by setting the instruction constant k appropriately. This is the induction step for the assignment statement, with the inductive result for pure expressions as hypothesis, and it implies side-effecting expressions are compiled both to preserve the intended 'nominal value' per (8c) and to preserve principle ($\widetilde{\mathbb{h}}$).

4.3 Difference Calculus

A Hoare-style deterministic pre-/post-condition calculus [15] is a natural stepping stone to a stochastic calculus. The Hoare-style calculus expresses the evolution of the current difference scheme section during a compiler pass. The operational semantics of the code is not at issue, freedom of choice is.

Assignment. Generalise the x+y with intermediates in r_1, r_2 of 4.1 to e with intermediates in ρ=$\{r_0, \ldots, r_n\}$. The result z is stored in r_0. The delta offsets have value Δr_i in r_i before and value $\Delta' r_i$ after the assignment. That is:

$$\begin{array}{c} \{\Delta r_0,\ \Delta r_1,\ \ldots,\ \Delta r_n\} \\ \mathsf{z} = e \\ \{\Delta' r_0,\ \Delta' r_1,\ \ldots,\ \Delta' r_n\} \end{array} \tag{9}$$

By (6b,8b), $\Delta' r, \Delta r$ can be independently chosen. Reading the Δ as a vector:

$$\{\Delta\}\ \mathsf{z} = e\ \{\Delta'\} \tag{9a}$$

$$\Delta \supseteq \Delta'|_{\bar{\rho}} \tag{9b}$$

That is, Δ extends $\Delta'|_{\bar{\rho}}$, Δ (possibly) differs from Δ' on ρ and is unaltered from it on the *complement* $\bar{\rho}$ of ρ. The Δ are indexed by the full range of registers and memory locations but in practice only a small subset need be considered.

When pointers (memory addresses calculated dynamically) are available to programmers, the type system of the source language must be augmented so each pointer is declared as pointing into a named global array as workspace:

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

That limits the possible memory locations (indices of Δ) for ptr to A. An unrestricted pointer may gain any address at runtime, which results in the compiler producing impossibly large/slow code, so the programmer wants to use **restrict**.

Conditionals. An if then else is compiled to machine code using branch instructions, but which branch is for true and which for false is varied by the compiler. It randomly chooses to generate code for b or for $\neg b$ at each level of boolean subexpression. The compile procedure is detailed in [4], but it has been described here already: the result b of each boolean subexpression is modified by a randomly chosen 1-bit delta δ to $b + \delta \mod 2$ just as for arithmetic expressions except that the arithmetic is 1-bit (i.e., mod 2), not 32-bit (mod 2^{32}).

The same technique is used in classic 'garbled circuits' [22] technology for obfuscating hardware logic circuit design – an arbitrarily selected exclusive-or (i.e., addition mod 2) mask is applied to inputs and outputs of every gate in the circuit in order to recover the designer's intended logic.

The compiler ensures that whichever branch is taken at runtime, the same difference scheme avails for the instruction after the conditional. It appends **add** instructions at the end of each branch as necessary for that. The upshot is the logic is nondeterministic choice. Let ρ be the registers written in e. The rule is:

$$\frac{\{\Delta_1\}\ s_1\ \{\Delta'\} \quad \{\Delta_2\}\ s_2\ \{\Delta'\}}{\{\Delta\}\ \mathsf{if}\ \ (e)\ s_1\ \ \mathsf{else}\ \ s_2\ \{\Delta'\}} \tag{10a}$$

$$\Delta \supseteq \Delta_1|_{\bar{\rho}} \cup \Delta_2|_{\bar{\rho}} \tag{10b}$$

and Δ_1, Δ_2 are equal on $\bar{\rho}$ after e, independently chosen on ρ by the compiler, and deltas Δ' are set up by it to be equal at the end of both branches, per ($\underset{\sim}{\mathbb{h}}$).

Loops. The compiler implements do while loops as code for the body followed by a conditional branch back to the start. Let ρ be the registers written in e and put $\Delta_1|_{\bar\rho}{=}\Delta_2|_{\bar\rho}{=}\Delta'|_{\bar\rho}$ in (10a),(10b) to get the following rule for compiled code:

$$\frac{\{\Delta\}\ s\ \{\Delta'\}}{\{\Delta\}\ \ \mathsf{do}\ \ s\ \mathsf{while}\ \ e\ \{\Delta'\}} \tag{11a}$$

$$\Delta \supseteq \Delta'|_{\bar\rho} \tag{11b}$$

The compiler sets deltas $\Delta|_\rho$, $\Delta'|_\rho$ independently. Per ($\widetilde{\mathbb{h}}$), the deltas are arranged to be the same values $\Delta'|_{\bar\rho}$ at beginning and end of the loop.

4.4 Calculus of Chaos

Let f_r be the probability distribution of offset Δr from nominal value v in register r, as the compilation varies stochastically. That is prob$[s_r = v{+}d] =$ prob$[\Delta r{=}d] = f_r(d)$, where s is the processor state. A stochastic analogue (12) of (9) is obtained by regarding each Δr, $\Delta' r$ as a random variable. Let variable x be stored in location $r_\mathsf{x} = L\mathsf{x}$, y in $r_\mathsf{y} = L\mathsf{y}$, z in $r_\mathsf{z} = L\mathsf{z}$. Then:

$$\begin{array}{c} \{\Delta r_\mathsf{x},\ \Delta r_\mathsf{y},\ \Delta r_\mathsf{z}\} \\ \mathsf{z} = \mathsf{x} + \mathsf{y} \\ \{\Delta' r_\mathsf{x},\ \Delta' r_\mathsf{y},\ \Delta' r_\mathsf{z}\} \end{array} \tag{12}$$

That asserts possibly different probability distributions before and after the assignment. Now let T be the run-time trace of a program. That is a sequence consisting of instructions executed and the values each read and wrote.

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 longer T' (it cannot decrease) is interpretable as the number of bits of unpredictable information introduced. These two facts from information theory will be needed:

Fact 1 *The flat distribution $f_X{=}1/k$ constant is the unique one with maximal entropy $\mathrm{H}(X){=}\log_2 k$, on a signal X that can take k values.*

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

Fact 1 identifies maximal entropy as n on n-bit space, achieved when each of the 2^n possible values is equiprobable. That is a disordered, i.e., 'chaotic', signal. Fact 2 uses the result (Shannon [19]) that the entropy of the sum of two n-bit signals is no less than that of either. The inference is that the characteristics of any distribution on a finite point space are obscured completely, not partially, by adding a 'chaotic' signal to it, i.e., one with flat, uniform distribution.

Below, logic is given for this stochastic view of compilation for the three source code constructs treated in 4.3, supposing the compiler implements ($\widetilde{\mathbb{h}}$).

Assignment. As in (9a), for pre-/post-condition:

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

but the Δ, Δ' are vectors of random variables Δr, $\Delta' r$. Let $\rho{=}\{r_0, \ldots, r_n\}$ be the registers written in e or in writing to z. For $r \notin \rho$, $\Delta' r{=}\Delta r$, as those r are unchanged, by (9b), so the same condition $\Delta|_{\bar\rho}{=}\Delta'|_{\bar\rho}$ holds here. I.e.:

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

We suppose the compiler follows ($\widetilde{\mathbb{h}}$), and that means each new random variable is independent with maximal entropy and each represents the compiler's free choice of embedded constant, like k of (6a,8a), in 'an arithmetic instruction that writes'. Then the machine code instruction that writes r_{z} does so with a delta that is a uniformly distributed independent random variable U and that increases the trace entropy to $\mathrm{H}(T'){=}\mathrm{H}(T){+}\mathrm{H}(U)$. The delta is 32-bit on a 32-bit platform, chosen with flat distribution by the compiler, per ($\widetilde{\mathbb{h}}$), so $\mathrm{H}(U){=}32$. There are $n{+}1$ registers $r_0, \ldots, r_n$ written independently, including that for z, so trace entropy increases by $32(n{+}1)$ bits. For any predicate $p(x)$, e.g., $h = x$:

$$\{p(\mathrm{H}(T){+}32(n{+}1))\}\ \ \mathsf{z} = e\ \ \{p(\mathrm{H}(T'))\} \tag{13c}$$

If the machine code instruction that writes has appeared earlier in the trace, the delta is already known, and the increment in trace entropy is zero second time:

$$\{p(\mathrm{H}(T))\}\ \ \mathsf{z} = e\ \ \{p(\mathrm{H}(T'))\} \tag{13c0}$$

Conditionals. As in (9b),(10b) but with random variables for the deltas:

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

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

The deltas $\Delta r{=}\Delta_1 r{=}\Delta_2 r$ are all the same for $r{\notin}\rho$ by (10b). 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 branch s_1 or s_2:

$$\frac{\{p(\mathrm{H}(T'))\}\, s_1\, \{q\}\quad \{p(\mathrm{H}(T'))\}\, s_2\, \{q\}}{\{p(\mathrm{H}(T){+}32n)\}\ \ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \ \{q\}} \tag{14c}$$

The compiler inserts extra 'arithmetic instructions that write' (**add**s) so the entropy increase is the same in both branches. It can because, even for loops, the entropy increase is finite and bounded (see below).

If the conditional appears in the trace a second time and branches the same way again then that contributes zero entropy as the deltas are already known:

$$\{p(\mathrm{H}(T))\}\ \ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \ \{p(\mathrm{H}(T'))\} \tag{14c0}$$

If it branches differently from the first time, the branch (but not the test) contributes entropy, as the deltas in that branch are yet unknown. But the, say m, instructions that align final deltas are constrained in (10b) to agree with the deltas in the other branch, which are already known. So those m do not count:

$$\frac{\{p(\mathrm{H}(T'))\}\ s_1\ \{q\} \quad \{p(\mathrm{H}(T'))\}\ s_2\ \{q\}}{\{p(\mathrm{H}(T)+32m)\}\ \mathsf{if}\ \ (b)\ s_1\ \ \mathsf{else}\ \ s_2\ \{q\}} \tag{14c1}$$

Those m instructions that 'align final deltas' with the other branch have a name:

Definition 2 *An instruction emitted by the compiler to adjust a final delta to agree with that in a joining control path is called a* trailer *instruction.*

Loops. Let $\rho=\{r_1,\ldots,r_n\}$ be the registers written during b. Then, per (11a), (11b), but with random variables as deltas, the following rule holds:

$$\frac{\{\Delta\}\ s\ \{\Delta'\}}{\{\Delta\}\ \ \mathsf{do}\ \ s\ \mathsf{while}\ \ (b)\ \{\Delta'\}} \tag{15a}$$

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

That means $\Delta r=\Delta' r$ for $r\notin\rho$. Those distributions are equal because the values are equal, by (13b), and trailer instructions reestablish Δ on the loop back-path.

A trace over the loop is always the same length between recompiled codes, because the compiler varies data values, not the semantics at a deeper level (see conserved nominal values in 4.1). 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 s and N instances of the loop test b. The variation in the trace is only that of (a) s repeated once, because the same m deltas appear second time too, and (b) b repeated once, for the same reason, with n deltas. The entropy calculation is (a) plus (b), no matter what $N\geq 1$ is (a $\mathsf{do}\ \ \mathsf{while}\ \ $ loop repeats at least once):

$$\frac{\{p(\mathrm{H}(T)+32m)\}\ s\ \{p(\mathrm{H}(T'))\}}{\{p(\mathrm{H}(T)+32(n+m))\}\ \ \mathsf{do}\ \ s\,\mathsf{while}\ \ b\ \{p(\mathrm{H}(T'))\}} \tag{15c}$$

So **do while** lengthens the trace like a loop but adds entropy to it like a conditional. Note that second time through the loop, zero entropy is added, because the same deltas are repeated:

$$\{p(\mathrm{H}(T))\} \quad \mathsf{do}\ \ s\ \mathsf{while}\ \ b\ \ \{p(\mathrm{H}(T'))\} \tag{15c0}$$

Equations (13c),(13c0),(14c),(14c0),(14c1),(15c),(15c0) are an information entropy calculus for runtime traces when compilation follows ($\widetilde{\mathbb{h}}$). Counting up:

Lemma 1 *The entropy of a trace is* $32(n+i)$ *bits where* n *is the number in it of distinct arithmetic instructions that write (a pair of trailer instructions that regulate the same delta count as one and a trailer instruction that reestablishes an earlier delta does not count) and* i *is the number of inputs.*

'Inputs' are those instructions that read a location that has not been written. (*Remark*) The logic holds for incomplete and/or non-contiguous sub-traces too.

The following characterises the compiler strategy that produces the maximum run-time trace entropy:

Proposition 1 *The entropy in the run-time traces induced by a compiler following the principle* ($\widetilde{\mathbb{h}}$) *as modified by* ($\underset{\sim}{\mathbb{h}}$) *is maximal among compositional compilation strategies.*

Proof: The issue is over whether a compiler could put more entropy into the run-time trace. The final deltas for data that is not read by following code do not have to be in agreement along both branches of a conditional, for example, so not following ($\underset{\sim}{\mathbb{h}}$) for them does no harm. But a compiler that works compositionally does not know the eventual context in which the code will be used so it must suppose that data that is written will later be read, and so must arrange for agreement between all final offset deltas in both branches of conditionals, enforcing ($\underset{\sim}{\mathbb{h}}$) in that case, indeed in all cases.

The only other way to put more entropy into the trace is to vary individual instructions more, but that is impossible if the compiler implements ($\widetilde{\mathbb{h}}$). ■

That characterisation decides details of chaotic compilation. For example, to the question of whether an array should have (a) one delta common to the whole array or (b) one per entry, the answer is (b) one per entry. One per array would mean each write to an entry must be followed by a 'write storm' to all other entries too, to realign their deltas to the newly written entry's (which is changed because the write instruction must add entropy to the trace). But the write storm's write instructions import no entropy as their deltas are all the same as the first, contradicting the characterisation.

The Proposition implies that, on a 32-bit platform, 32 bits of entropy per datum are provided by a chaotic compiler, a (weak) form of semantic security:

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

The general interest is with multiple data values observed at different points in the trace. The result depends on how they are connected computationally:

Definition 3 *Two data values in the trace are (entropy)* 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 m independent points, the variation obtained by a chaotic compiler is maximal, i.e., $32m$ bits, because the data is not constrained by ($\underset{\sim}{\mathbb{h}}$):

Theorem 1 *The probability across different compilations that any m particular 32-bit values beneath the encryption in the trace are precisely x_i, provided they are pairwise independent, is* $1/2^{32m}$.

(*Remark*) Each dependent pair reduces the entropy by 32 bits.

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 reference

5 Implementation

Our prototype 'chaotic' compiler `http://anonymised.url` is for ANSI C [17], 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**). 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[7] of the Ackermann function[8] [20] is shown in Table 2, with null encryption for better visibility. The instruction constants and values written to registers are encrypted on an encrypted computing platform, with, e.g., $\mathcal{E}$[`-86921031`] in place of `-86921031`.

6 (Informal) Security Argument

Here is a sketch proof that a chaotic compiler makes programs 'safe from polynomial-time discovery' of what the data in the runtime-trace is intended to mean, in the context of encrypted computing. The claim is that there is no polynomial time method M that can estimate the value of a designated bit b in the trace of a program P, if there is none that succeeds against the encryption alone. Success means with a probability that exceeds $1/2$ by some margin $B > 0$ infinitely often as the word size n tends to infinity, but the precise notion may be varied for the proof: for example, being correct about b with probability $p > 1/2 + 1/n$.

[7] For readability here, the final delta in register **v0** is set to zero.

[8] 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)); }.

Proof: [Sketch] Let the compiler unroll source code loops to depth $N{=}2^n$ and inline function calls to depth N and push code after conditional blocks into both branches to depth N, leaving no branches, loops or function calls for N steps. By Theorem 1 a chaotic compiler generates object code for P with maximal entropy in at least the first N instruction cycles of the run-time traces, measured from one recompilation to the next, following ($\widetilde{\mathbb{h}}$). The constraint ($\underset{\sim}{\mathbb{h}}$) does not apply.

By Theorem 1, there is no algebraic or any other relation M can rely on among the $m{\leq}p(n){\leq}N$ trace-points it has time to access, for polynomial p of order k, and M must depend on its capability against the encryption alone, which it is hypothesised to be not successful against. ■

The same argument works for any number of bits.

7 Conclusion

In summary, this paper discusses stochastic compilation and defines chaotic compilation as stochastic compilation with maximum entropy. The compiler works with a difference scheme describing the variation from nominal of the value in each register and memory location, differing per instruction in the machine code program. A program logic of differences extends to an information entropy calculus for run-time traces that quantifies chaotic compilation. That feeds an argument for security against polynomial-time complexity methods of attack against encrypted computing. The unsual aspect here is software engineering in the cause of mathematics. Definition and proof of security for encrypted computing has been the goal, and the idea of a chaotic compiler is to allow mathematical reasoning for the stochastic properties to be replaced by engineering for them.

The chaotic C compiler ('havoc') is available from the open source project at `http://sf.net/p/obfusc` and covers all of ANSI C except longjmp/setjmp. Array access is $\mathbf{O}n$ but otherwise the compiled code is not slower than normal.

Acknowledgments: Simon Pickin's work has been supported by the Spanish MINECO-FEDER (grant numbers DArDOS, TIN2015-65845-C3-1-R and FAME, RTI2018-093608-B-C31). Peter Breuer thanks Hecusys LLC for continued support in encrypted computing research, and London South Bank University for support via a visiting research fellowship.

References

1. Breuer, P., Bowen, J.: A fully homomorphic crypto-processor design: Correctness of a secret computer. In: Proc. Int. Symp. Eng. Sec. Softw. Sys. (ESSoS'13), pp. 123–38. No. 7781 in LNCS, Springer (2013). https://doi.org/10.1007/978-3-642-36563-8_9
2. Breuer, P., Bowen, J.: A fully encrypted high-speed microprocessor architecture: The secret computer in simulation. Int. J. Crit. Computer-Based Sys. **9**(1/2), 26–55 (2019). https://doi.org/10.1504/IJCCBS.2019.10020015
3. Breuer, P., Bowen, J., Palomar, E., Liu, Z.: A practical encrypted microprocessor. In: Callegari, C., et al. (eds.) Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16). vol. 4, pp. 239–50. SCITEPRESS, Port. (Jul 2016). https://doi.org/10.5220/0005955902390250

4. Breuer, P., Bowen, J., Palomar, E., Liu, Z.: On obfuscating compilation for encrypted computing. In: Samarati, P., et al. (eds.) Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17). pp. 247–54. SCITEPRESS, Port. (Jul 2017). https://doi.org/10.5220/0006394002470254
5. Breuer, P., Bowen, J., Palomar, E., Liu, Z.: On security in encrypted computing. In: Naccache, D., et al. (eds.) Proc. 20th Int. Conf. Info. Comm. Sec. (ICICS'18), chap. 12, pp. 192–211. No. 11149 in LNCS, Springer, Cham. (Oct 2018). https://doi.org/10.1007/978-3-030-01950-1_12
6. Breuer, P., Bowen, J., Palomar, E., Liu, Z.: Superscalar encrypted RISC: The measure of a secret computer. In: Proc. 17th Int. Conf. Trust, Sec. Priv. Comp. Comms. (TrustCom'18). pp. 1336–41. IEEE Comp. Soc. (2018). https://doi.org/10.1109/TrustCom/BigDataSE.2018.00184
7. Conway, J.H.: Fractran: A simple universal programming language for arithmetic. In: Cover, T.M., Gopinath, B. (eds.) Open Problems Commun. Comp., pp. 4–26. Springer, Heidelberg/Berlin (1987). https://doi.org/10.1007/978-1-4612-4808-8_2
8. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Proc. 29th Int. Conf. Th. Appl. Crypto. Tech. (EUROCRYPT'10). pp. 24–43. Springer (May 2010)
9. Fletcher, C.W., van Dijk, M., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: Proc. 7th ACM Work. Scal. Trust. Comp. (STC'12). pp. 3–8. ACM, NY, USA (2012)
10. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proc. 41st Ann. ACM Symp. Th. Comp. (STOC'09). pp. 169–178. NY, USA (2009)
11. Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) Proc. 30th Int. Conf. Th. Appl. Crypto. Tech. (EUROCRYPT'11), pp. 129–148. No. 6632 in LNCS, Springer (2011)
12. Goldwasser, S., Micali, S.: Probabilistic encryption & how to play mental poker keeping secret all partial information. In: Proc. Ann. ACM Symp. Th. Comp. pp. 365–77. (STOC'82), ACM (1982)
13. Goldwasser, S., Micali, S.: Probabilistic encryption. J. Comp. Sys. Sci. **28**, 270–299 (1984)
14. den Hartog, J.I.: Verifying probabilistic programs using a Hoare-like logic. In: P.S., T., R., Y. (eds.) Proc. 5th Annual Asian Computing Science Conference (ASIAN'99). LNCS, vol. 1742, pp. 113–125. Springer, Berlin/Heidelberg (1999). https://doi.org/10.1007/3-540-46674-6_11
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–80 (1969). https://doi.org/10.1145/363235.363259
16. Irena, F., Murphy, D., Parameswaran, S.: CryptoBlaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support. In: Proc. 23rd Asia S. Pac. Des. Auto. Conf. (ASP-DAC'18). pp. 702–8. IEEE (2018)
17. ISO/IEC: Programming languages – C. 9899:201x Tech. Rep. n1570, Int. Org. for Standardization (Aug 2011), JTC 1, SC 22, WG 14
18. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Prog. Lang. Sys. (TOPLAS) **18**(3), 325–353 (1996). https://doi.org/10.1145/229542.229547
19. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal **27**(3), 379–423 (Oct 1948). https://doi.org/10.1002/j.1538-7305.1948.tb01338.x
20. Sundblad, Y.: The Ackermann function: a theoretical, computational, and formula manipulative study. BIT Num. Math. **11**(1), 107–19 (1971)
21. Tsoutsos, N.G., Maniatakos, M.: The HEROIC framework: Encrypted computation without shared keys. IEEE TCAD IC Sys. **34**(6), 875–88 (2015)
22. Yao, A.C.C.: How to generate and exchange secrets. In: 27th Ann. Symp. Found. Comp. Sci. pp. 162–167. IEEE (1986)