Chaotic Compilation for Encrypted Computing: Obfuscation but Not in Name

Peter T. Breuer

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

Abstract. An 'obfuscation' for encrypted computing is quantified exactly here, leading to an argument that security against polynomial-time attacks has been achieved for user data via the deliberately 'chaotic' compilation required for security properties in that environment. Encrypted computing is the emerging science and technology of processors that take encrypted inputs to encrypted outputs via encrypted intermediate values (at nearly conventional speeds). The aim is to make user data in general-purpose computing secure against the operator and operating system as potential adversaries. A stumbling block has always been that memory addresses are data and good encryption means the encrypted value varies randomly, and that makes hitting any target in memory problematic without address decryption, but decryption anywhere on the memory path would open up many easily exploitable vulnerabilities. This paper 'solves (chaotic) compilation' for processors without address decryption, covering all of ANSI C while satisfying the required security properties and opening up encrypted computing for the standard software tool-chain and infrastructure. That produces the argument referred to above, which may even hold without encryption.

Keywords: Obfuscation · Compilation · Encrypted computing

1 Introduction

This article explains recent advances in understanding and practice in the emerging technology of *encrypted computing* [1–6], in particular 'chaotic compilation' for this context [7], which supports the security proofs and now has broken through to allow essentially all extant ANSI C [8] source codes to be compiled (the known lacks are longjmp/setjmp), nearly as they stand (strict typing is necessary), opening up the field. An unexpected by-product is an argument that obfuscation safe against polynomial time attacks is produced by this kind of compilation, perhaps even in the absence of encryption.

Encrypted computing means running on a processor that takes encrypted inputs to encrypted outputs via encrypted intermediate values. The operator and operating system are the potential adversaries in this context. Encrypted computing aims to:

Protect user data from the powerful operator of the system.

A subverted operating system is as much 'the operator' as is a human with administrative privileges, perhaps obtained by interfering with the boot process, and this document will use 'the operator' indiscriminately for both. From a systems perspective, the operator (or operating system) are merely the operator mode of the processor. A processor starts in operator mode when it is switched on, in order to load operating system code into reserved areas of memory from disk. Conventional application software relies on the processor to change from user mode to operator mode and back again for system support (e.g., disk I/O) as required, so the operator mode of the processor presents difficulties as adversary. Security is possible only because 'an honest operator should not care what user data means'

– informally, the operator/system is there to change the tapes when the machine beeps.

The reader should take onward with them from the outset that a processor for encrypted computing works encrypted only in user mode. In operator mode it works unencrypted (with unrestricted access to registers and memory) entirely as usual. A good mnemonic is:

Privilege means no encryption, no privilege means encryption.

Any user data encountered while in operator mode (and also in user mode) will be in encrypted form so the operator's privilege of unrestricted access does not necessarily imply understanding of what user data means, or an ability to alter it meaningfully. It can always be copied for later analysis, or overwritten with zeros, one word substituted with another, etc, but security proofs [9] show that cannot amount to more than guesswork.

A fundamental technical motivation in this area is the intuition that a processor that 'works encrypted' must inherently be not less secure than one that does not, and there are reasons why it should not be (much) slower that have been borne out in hardware, as described below. The non-technical motivations are manifold and should not need enumeration. One plausible scenario for an attack by the operator is where cinematographic animation is rendered in a server farm. The computer operators have an opportunity to pirate for profit parts of the movie before release and they may be tempted. Another possible scenario is where a specialized facility processes satellite photos of a foreign power's military installations to spot changes. If an operator (or hacked operating system) can modify the data to show no change where there has been some, then that is an option for espionage. Informally:

Definition 1 (Successful attack)**.** *A* successful attack *by the operator is one that discovers the plaintext of user data or alters it to order.*

That is meant statistically, so an attack that succeeds more often than chance also counts.

Overwriting encrypted data with zeros does not count because that does not set the plaintext under the encryption to a value that the adversary can predict, even stochastically, supposing the encryption itself is secure. The operator can always act chaotically or destructively, by rewriting memory randomly or turning the machine off, for example.

Pointers (and arrays) are the major obstacle to compiling real, extant, source codes in the encrypted computing setting. Pointers underly any practical programming language: in C (and C++) they are explicit, but in other languages they are implicit. In Fortran 95 all variables are implicitly pointers, so passing the variable to a subroutine allows the caller's variable to be changed by the callee. Passing an array allows entries in the caller's array to be rewritten in most languages. In Java every object is a pointer, and a new object instead of another pointer to the same object must be created via **clone**().

'The problem with pointers' is that (1) the memory unit will receive an encrypted address ('pointer') and (2) it is not privy to the encryption. The rationale for (1) is that while the processor design could accommodate address decryption on the memory path, that opens up an attack vector in which the operator passes encrypted data as an address in a load or store machine code instruction and checks (physically or programmatically) where in memory it accesses, thus decrypting the data, so it is not a good idea for security. It is also faster for hardware not to have encryption or decryption units on the memory path. The rationale for (2) is an attacker with physical access could walk away with the RAM chips and they might contain plaintext or keys if the memory unit could do decryption itself (c.f., *cold boot* attacks [10,11], physically freezing RAM to keep data while the chips are moved), or if the processor hardware decrypts for it.

The conclusion to draw is that memory should do without decryption both internally and externally. But then it will receive some ciphertexts that are each a different encryption of the same programmed address and it cannot tell they are intended to mean the same thing: the programmer says to write x at address 123 and the encrypted value 0x123456789A of

123 is passed to the memory as the ciphertext for the address at runtime; if the programmer aspires to read it back via address 123 then a ciphertext 0xA987654321 that is an alternative encryption of 123 may be passed as address, and content y different to x is returned from there. One can get around that in hand-crafted fashion by specifying exactly every time what encrypted address is to be used, but that only works for simple programs – it will break down for programs that use arrays of pointers to other arrays of pointers to objects which may contain pointers. In other words, programs will break, often silently. Strict typing can limit programs to those that will work in the setting, but the programming experience would be frustrating and porting existing programs to the platform would require a prohibitively high degree of semantic understanding of the original.

We have now learned how to work around this hardware semantics problem via software logic, while maintaining the security properties required. That opens up encrypted computing for a traditional software development cycle involving high-level source language programming and a compile, assemble, link, load tool-chain with operating system support.

This article is organized as follows. Section 2 will set out bullet points about encrypted computing platforms to refer to in the rest of the text. The reader should defer to those points. For example, it is not the case that the operator has access to the encryption key, as a reader may suppose from experience with conventional platforms, and the first point emphasizes that. Nor is encryption done in software, as the reader may have experienced, so it does not take different times for different data. Nor do instructions aborted while still in the processor pipeline leave a mark in cache, nor may they otherwise be detected, etc. There is nearly a decade of research in the field, and there are no easily picked openings. Section 3 expresses abstract terms that hopefully resonate with cryptologists, but ready-made matches for this new field are scarce and the reader should first and foremost take the concepts defined there on their own terms. Section 4 describes existing platforms for encrypted computing and discusses the theory. Section 5 resumes a modified OpenRISC (`http://openrisc.io`) machine code instruction set for encrypted computing described in [7], needed for security. Section 6 first resumes the basis of chaotic compilation restricted to integers and call by value a la [7] and then covers ramified atomic types (long integers, floats, doubles, etc.), arrays and pointers (hence call by reference), 'struct' (record) and union types. Theory is developed in Section 7. It quantifies the entropy in a runtime trace for code compiled following ($\mathcal{T}$), characterizing that as 'best possible'. Section 8 discusses further implications for security, including the argument that this setting has security against polynomial time (in the word-length, n) attacks on user data, with or without encryption.

2 Reference Points

This section sets out touchstones on encrypted computing for the reader to refer to.

§2.1 **Encryption key access** is impossible programmatically. The key is embedded in the processor hardware and there are no instructions that read it. Keys are either installed at manufacture, as with Smartcards [12], or uploaded securely in public view via a Diffie-Hellman hardware circuit [13] to a write-only and otherwise inaccessible internal store. The privy user (which term does not include the operator) can always interpret the encrypted data, elsewhere, for safety, as they know what the key is.

§2.2 **Prototype processors** for encrypted computing at near conventional speeds already exist (see Section 4) and have existed since a first recognizable try at the idea [1] about 2012. Before that, computing *systems* (not processors) that operated partially encrypted were not uncommon for commercial movie and music reproduction going back as far as 1993 [14] and 2000 [15], but their processor component operated in the ordinary way while memory (RAM) or disk or other peripherals stored in encrypted form the media to be safeguarded. Modern Intel SGX™ systems [16–18] are not conceptually different in that regard, but with extra sandbox technology (memory enclaves into which the programmer may voluntarily consign their code, with separate caches and register sets). The idea in encrypted computing is to rely instead on encryption through-and-through as the security barrier protecting user data. A basic question to answer is whether encrypted computing potentially reduces the security of the encryption ('no', per [9]).

§2.3 **Platform/hardware issues** such as the real randomness of random numbers or power side-channel information leaks [19] will not be at issue here. Existing technological defenses for conventional processors [20] may be applied in practice, and are to be supposed.

§2.4 **The intended mode of working** is 'remote processing':

The user compiles program and data, sends both away, and gets back output.

New data means a new encoding and compilation. The word 'encoding' is used because more than encryption (and more than vanilla compilation) is involved. The compilation is 'chaotic', as explained below. That involves producing both a new encoding for the data throughout the program and a new machine code program that can cope with it and then encoding and encrypting the data and partially encrypting the machine code (only constants in the instructions are encrypted - the 'opcode' and other fields are plaintext).

Continuous running may become feasible as technology for dynamic code update [21] in this setting develops, but for the purposes of this paper a mode of use other than the above is not contemplated.

§2.5 **Key management** is not an issue via this argument: if (a) user B's key is still loaded when user A runs, then A's programs do not run correctly as the encryption is wrong for them; if (b) B's key is in the machine together with B's program when A runs, then user A cannot supply appropriate encrypted inputs nor interpret the encrypted output.

§2.6 **Security** user on user boils down in this setting to security for the user against the operator as the most powerful potential adversary in the system, and it is proved in [9] (the argument is reprised in the Appendix; Lemma A1), that

(i) a processor that supports encrypted computing,
(ii) an appropriate machine code instruction set architecture,
(iii) a 'chaotic' compiler as described below

together provide a property that parallels for this setting classic *cryptographic semantic security* [22] (CSS) for encryption, better known via the semantically equivalent *ciphertext indistinguishability under chosen-plaintext attack* (IND-CPA). The latter means there is no (polynomial time) method for an adversary to tell which is which between the ciphertexts of two plaintexts (stipulated by the adversary, both the same length), to any degree significantly above chance (as the key/block size tends to infinity).

§2.7 **An appropriate machine code instruction set** as referred to above is one that satisfies four axioms introduced in [5] and set out in Box II here. They are essentially (a) atomicity, (b) encrypted working, (c) malleability, and (d) absence of collisions between the ciphertext constants in program instructions and data at runtime.

§2.8 **Source code language coverage** for encrypted computing that extended to (32-bit plaintext) integers, arithmetic, conditionals and call-by-value was first achieved in [7] and is explained in Section 6. It is extended here to pointers and all of ANSI C [8] (except setjmp/longjmp), with its long long, float, double, array, struct (record) and union data types. That 'solves' the practical problem of compilation for all encrypted computing.

§2.9 **Primitive operations** in the processor will be taken to include all encrypted 32-bit integer arithmetic. That means for example that encrypted data can be 'added' by the hardware in one operation, via an appropriate machine code instruction, producing an encryption of $x + y \pmod{2^{32}}$ as result from encryptions of x and y as operands. Similarly for the other primitives of the computer arithmetic, which are specified exactly in Section 5. Careful specification is necessary because it turns out that the standard arithmetic operations in their conventional form are dangerous to security [23].

Since hardware is not the focus of this paper, encrypted 64-bit integer arithmetic will also be taken as primitive. It is carried out on two encrypted 32-bit integers representing the high and low bits respectively (software subroutines for each operation using the 32-bit instruction set only is an alternative). Encrypted (32-bit and 64-bit) floating point arithmetic will only be treated in the Appendix but should also be taken as primitive. These primitives are each supported by at least one of the prototype processors discussed in Section 4.

Notation

Encryption (with key K understood) will be denoted $x^{\mathcal{E}}$ or $\mathcal{E}[x]$ of plaintext value x and should be read as a multi-valued function of x, i.e., a single-valued function $\mathcal{E}[p \cdot x]$ when the hidden padding p is taken into account. Decryption is $\zeta^{\mathcal{D}} = \mathcal{D}[\zeta]$, with $x = \mathcal{D}[x^{\mathcal{E}}]$. The encryption aliases of a ciphertext ζ_0 are those ζ_1 with the same plaintext as it under decryption $\mathcal{D}$, i.e., $\zeta_0 \underset{\mathcal{D}}{\equiv} \zeta_1$, meaning $\mathcal{D}[\zeta_0] = \mathcal{D}[\zeta_1]$. To avoid excess notation that equivalence is written as *equality* on the cipherspace, so equal aliases $\zeta_0 = \zeta_1$, i.e., $\zeta_0 \underset{\mathcal{D}}{\equiv} \zeta_1$, may be non-identical, with $\zeta_0 \underset{\text{id}}{\neq} \zeta_1$. They are encryption alternatives or *aliases* of the same plaintext $x = \zeta_0^{\mathcal{D}} = \zeta_1^{\mathcal{D}}$. Then $x^{\mathcal{E}}$, $\mathcal{E}[x]$ denote particular but unspecified aliases, and $\zeta = \mathcal{E}[\mathcal{D}[\zeta]]$ is true (recall that equality on the cipherspace is equivalence under decryption). Key K may be mentioned as an extra parameter in $\mathcal{E}[K, x]$ and $\mathcal{D}[K, \zeta]$. Padding varies the value but not the $\underset{\mathcal{D}}{\equiv}$ equivalence class, which is the equality, with $\mathcal{E}[p \cdot x] = \mathcal{E}[q \cdot x] \underset{\text{id}}{\neq} \mathcal{E}[p \cdot x]$ for $p \neq q$, and $\mathcal{E}[p \cdot x] \neq \mathcal{E}[p \cdot y]$ for $\mathcal{E}[x] \neq \mathcal{E}[y]$.

The operation on the ciphertext domain corresponding to o on the plaintext domain will be written $[o]$, where $x^{\mathcal{E}}\,[o]\,y^{\mathcal{E}} = \mathcal{E}[x\,o\,y]$. The relation on the ciphertext domain corresponding to R on the plaintext domain will be written $[R]$, where $x^{\mathcal{E}}\,[R]\,y^{\mathcal{E}}$ iff $x\,R\,y$. These are well-defined with respect to the cipherspace equality.

3 Key Security Concepts

The game-theoretic formulation of the classic IND-CPA security definition for this setting is:

G1.1 the operator selects any program p and input data d;
G1.2 the user compiles it (twice) and shows the operator the two plaintext codes p_1, p_2 and input data d_1, d_2;
G1.3 the user encrypts those to p_1', p_2' and d_1', d_2' and passes those to the encrypted computing platform;
G1.4 the operator examines the running codes $p_1'(d_1')$, $p_2'(d_2')$, interferes, experiments, observes intermediate and final results e_1', e_2' (all in encrypted form);
G1.5 the operator attempts to say which is which.

The argument of Lemma A1 shows that in an encrypted computing context there is no method (in particular no polynomial time one) the operator can use to be right more often than if they were trying plain IND-CPA against the encryption. The probability of success given that there is no advantage against the encryption alone is not different from chance, i.e. 1/2 (exactly).

Definition 2. That the operator cannot win the game above with different than 1/2 probability (exactly, for all lengths of the encrypted word on the platform) for this context, given that the encryption itself is independently secure against IND-CPA, will be called *cryptographic semantic security* relative *to the security of the encryption* (ρCSS), for user data against operator mode as adversary.

The rationale behind it is discussed below, but it is clear that it holds for certain programs p: those that have no instructions and thus do nothing to data, passing input to output directly, without any interference. Given that the encryption is secure, an adversary can do nothing but guess blindly as to what the data is.

An interesting corollary (Remark 2 of [9]) has it that there is an encrypted program that does decryption (it is the decryption algorithm for the encryption in use in the processor, compiled for the encrypted computing environment), but the operator cannot build it correctly, neither from scratch nor out of scavenged parts from encrypted programs, with any probability of success above random chance. That is so though they know what the encryption is (not the key) and the canonical structure of a program for decryption.

The property ρCSS is above all a statement about whether encrypted computing provides additional information to the operator that might lessen the security of the encryption ('no'). For example, it might be possible for the operator to recognize a computation $2+2=2*2$ (encrypted), via coincidence of the encrypted operands and results. The operator might try inserting the same encrypted value for (supposedly) 2 at all the places in that calculation and see the encrypted answers are identical. The argument of Lemma A1 says that no, even though the operator thinks they have identified the situation, in fact other interpretations of the operands and results that are consistent with the operational semantics of the processor are possible, and all interpretations are equally possible given the 'right' compiler as generator. That is a perspective from mathematical logic and model theory, and it may be expressed for a general reader as follows: if one takes the axioms like $x+y=y+x$ that describe the specially designed primitive operations of the processor (Table 1), and chooses facts $r_1=2$, $r_2=4$ etc. consistent with the axioms for the unknown values $r_1, \dots r_n$ at n chosen points in the runtime trace of a program, then there are also other, different but compatible choices such as $r_1=5$, $r_2=6$ etc. that are equally plausible. That holds for $n=1$ chosen point, wherever it is (Corollary 2 of [9]), and in general independently for any n chosen points, with certain exceptions characterized in this paper (for example, measuring the input r_1 and output r_2 of a copy instruction does not allow r_1 and r_2 to be chosen independently – one of them may be chosen independently, and then the other must be equal to it). An informal rendering of ρCSS might be that good security in encrypted computing boils down to good encryption.

A similar, but different, game is:

G2.1 the operator selects any program p and input data d and selects a polynomially defined point r_n in the trace (e.g., the last point at which register r changes in the first n^3 steps);
G2.2 the user compiles it for a platform that does n-bit computing beneath the encryption, forming p_n and d_n, which are not shared with the operator;
G2.3 the user encrypts to p'_n and d'_n and passes those to the encrypted computing platform;
G2.4 the operator examines the running code $p'_n(d'_n)$, interferes, experiments, observes intermediate and final results (all in encrypted form);
G2.5 the operator tries to say what the plaintext value beneath the encryption in r_n is.

The argument in Section 8 says that if the encryption is secure for IND-CPA, then the operator cannot win this game with a probabilty above chance, as $n \to \infty$ (but what the probability is for a given program p and word length n is not going to be known).

If the operator could win this game, then they could win the first game given at the front of this section (for sufficiently large n), so it ought not to happen. But the argument is independent and also suggests that encryption may not be necessary.

Chaotic compilation for encrypted computing is as follows. It is related to 'obfuscation' in that it makes code harder to read than it would otherwise be, but 'obfuscation' in the security field refers to transforming one source code to another, while compilation transforms source code to object code (assembler, or machine code), not source code. so a compiler cannot be a classical *obfuscator*. Instead, chaotic compilation has properties

Box I

(a) A fully homomorphic encryption (FHE) $\mathcal{E}$ of 1-bit data lacks the cryptographic semantic security (CSS) property.

$$\begin{aligned}\mathcal{E}[0]*\mathcal{E}[0]&=\mathcal{E}[0]\\ \mathcal{E}[0]*\mathcal{E}[1]&=\mathcal{E}[0]\\ \mathcal{E}[1]*\mathcal{E}[0]&=\mathcal{E}[0]\\ \mathcal{E}[1]*\mathcal{E}[1]&=\mathcal{E}[1]\end{aligned}$$

Guessing 0 as outcome is right 75% of the time.

(b) A FHE program that adds 2-bit data to itself:

$$\begin{aligned}\mathcal{E}[0]+\mathcal{E}[0]&=\mathcal{E}[0]\\ \mathcal{E}[1]+\mathcal{E}[1]&=\mathcal{E}[2]\\ \mathcal{E}[2]+\mathcal{E}[2]&=\mathcal{E}[0]\\ \mathcal{E}[3]+\mathcal{E}[3]&=\mathcal{E}[2]\end{aligned}$$

has output $y = 2x$ that is 100% even, breaking CSS.

required to prove ρCSS. It behaves stochastically: applied twice to the same source code, it emits two (probably) different object codes. Chaotic compilation always at least:

(A) *produces identically structured machine codes;*
(B) *produces identically structured runtime traces;*
(C) *varies only encrypted constants in the code;*
(D) *varies runtime data beneath the encryption*

across different recompilations of the same source code. The aim of (A-D) is that:

> *The object codes and their traces look the same to an ignorant observer.*

That is an observer ignorant of the encryption key and hence putatively unable to read the plaintext of encrypted words. The same instructions (modulo different embedded encrypted constants) will run in the same order, with the same branches and loops, but runtime data beneath the encryption differs following a scheme known only to the user. The math and computer science behind that is reprised in Section 4.

Chaotic compilation in particular means producing code within the framework of (A-D) such that an adversary cannot count on $0, 1, 2,$ etc. occurring frequently beneath the encryption in a program trace, which is naturally the case in a program written by a human. That would enable statistically-based dictionary attacks [24] against the encryption. The desired property is:

Definition 3 (Chaotic compilation)**.** No data value beneath the encryption appears at runtime with higher probability than another.

That is measured across recompilations, which are generated stochastically. There is no dependence on the key/block size n in that definition (which also makes it different in kind from classical definitions of obfuscation). The principle applies both for observations of single words and also for simultaneous ('vector') observations at multiple points in a trace, but that is much harder to achieve. This paper proves it for single words and quantifies it for vectors. Note that the property is literally violated, e.g, in implementations [25] of fully homomorphic encryptions [26,27] (FHE), where the output of a 1-bit AND (multiplication) operation is 0 beneath the encryption with probability $\frac{3}{4}$ (Box I (a)).[1]

What makes it hard to achieve Defn. 3 simultaneously for many different observed values at different points in the trace is that computational semantics has rules to it. Apart from (A-D), the 'chaotic compiler' is obliged to generate machine code in which:

[1]That 0 is a probable outcome from multiplication in a FHE $\mathcal{E}$ is not an extra liability because in 1-bit arithmetic $\mathcal{E}[x] + \mathcal{E}[x] = \mathcal{E}[0]$ with certainty from any observed encrypted value $\mathcal{E}[x]$. It can also be relied on that $\mathcal{E}[1]$ is one of the inputs in any nontrivial calculation because 'all-zeros' as inputs propagates through to all-zeros as output via $\mathcal{E}[0] + \mathcal{E}[0] = \mathcal{E}[0] * \mathcal{E}[0] = \mathcal{E}[0]$.

(E) *a copy instruction preserves data exactly;*
(F) *the variations introduced by the compiler are equal where any two control paths join.*

Any observer can deduce data beneath the encryption is copied in (E), because the same ciphertext is seen. The condition (F) refers to the end of a loop, after conditional blocks, at subroutine returns, at the label target of a **goto**. That it holds is deducible by an adversary from basic computational principles: since the number of times through a loop is generally not predictable at compile-time, the compiler must ensure the same conditions are re-established at the end of each loop as prevail at the start, ready for another go-through.

The constraints (E-F) impose an underlying order on what is desired to be an apparently chaotic scheme of variations from nominal in the plaintext data beneath the encryption in a program trace. An overall *strategy* for achieving Defn. 3) is as follows ($\mathcal{S}$):

Vary a machine code program's instruction constants so every possible data variation beneath the encryption is obtained with uniform *probability.* ($\mathcal{S}$)

The probability is taken across traces, one for each recompilation of the same source code. Unfortunately, modifications require knowledge of the programmer's intention as expressed in the source code, and reverse engineering that from machine code is in general a known Turing Halting Problem equivalent (i.e., computationally impossible). It is up to the compiler to provide variation, working stochastically from source code. However, the 'uniform ... across the range' is a key that indicates how the compiler ought to generate code that boils down to a particular tactic as explained below.

The compiler *tactic per increment of code* to implement the strategy ($\mathcal{S}$) is as follows:

Every machine code instruction that writes should introduce maximal entropy. ($\mathcal{T}$)

That refers to entropy introduced into the runtime trace by the compiler's variations.

What this means is that the compiler must exercise fully the possibilities for varying the trace at every opportunity in a compiled program. The only way to vary the trace is via an instruction that writes something, otherwise there is no effect. 'Writes' means any change that is testable, even if not observed directly – the outcome of a comparison instruction cannot be read from registers, for example, but it is tested by where the branch instruction jumps to, so it counts as a 'write'. The compiler should not, for example, always use 1 in an addition instruction if the possibility exists of using a different number and still satisfying the programmer's intention.

4 Background and Related Work

Several fast processors for encrypted computing are described in [28]. Those include the 32-bit KPU [5] with 128-bit AES encryption [29], which on the industry-standard Dhrystone [30] v2 benchmark is reported in [28] to run encrypted at the speed of a 433 MHz classic Pentium (tables equating different processors are at www.roylongbottom.org.uk/dhrystone%20results.htm) with a 1 GHz clock speed, and the older 16-bit HEROIC [3,4] with 2048-bit one-to-one Paillier encryption [31], which runs like a 25 KHz Pentium, as well as the recently announced 32-bit CryptoBlaze [32] with one-to-many Paillier. That is 10× faster than HEROIC (but branches are farmed out for decision to the remote user across the Internet and are not counted in that figure).

§4.1 **Ascend** [1] is retrospectively seen as the first exemplar of the class of processors designed for encrypted computing. This 32 (in plaintext) bit AES-based (co)processor aimed to work like a black box, literally: the processor was physically inaccessible and could not be interfered with programmatically until it had finished an execution task. It accepted encrypted program and data inputs and produced encrypted data outputs, and internals were unobservable. To obfuscate addressing to memory, oblivious RAM

(ORAM) [33] was integrated. Timing and power statistics in terms of what signals appeared on the data pins at what time were arranged to match specifications set beforehand. The machine code instruction set was MIPS RISC (i.e., uncomplicated – RISC stands for 'reduced instruction set computing' and MIPS (an autonym) is the variety taught in undergraduate courses) beneath the encryption. The operator was considered 'semi-honest', in contrast to the potential adversary of this paper. In common with the later designs, every instruction takes the same time and power to execute no matter what the data. As in reports on later projects (except [28]) speeds are not discussed except with respect to the same processor running without encryption, and a 12-13.5× slowdown is noted. The authors' reticence is likely because security audiences do not know that different instruction sets, processor architectures, platform technologies and compilers are incomparable, and would be nonplussed by a benchmark that equated to a 4 MHz Pentium, but is really remarkable. A 433 MHz Pentium benchmarks on Dhrystone at 114× one DEC VAX 11/780 minicomputer from 1977, and 4 MHz Pentium would 'wax the VAX.'

§4.2 **A modified arithmetic** is the principle behind later processors for encrypted computing and is known to be sufficient to generate encrypted working [2] since 2013. HEROIC and CryptoBlaze both embed the Paillier encryption, which is partially homomorphic, making the modified addition straightforward to do in hardware, with $x^{\mathcal{E}}\,[+]\,y^{\mathcal{E}} = \mathcal{E}[x+y] = x^{\mathcal{E}} * y^{\mathcal{E}} \mod m$. That is multiplication of the encrypted numbers modulo a 2048-bit integer m. HEROIC's 2048-bit ciphertext multiplication takes 4000 cycles of the supporting 200MHz hardware, so one encrypted (16-bit plaintext) addition takes 1/50,000s, comparable to a 25 KHz Pentium (one 32-bit addition every 1/25,000s).

The KPU uses AES, not a homomorphic encryption (it is reported as having been tried with Paillier, but the arithmetic proved impossible to pipeline advantageously), and gets its speed by decrypting internally once in the pipeline at the start of a sequential train of arithmetic micro-operations, taking one cycle each, and re-encrypting at the end. The overhead is reported as 10-20 cycles per train, integrated as 10 pipeline stages. That leads to approx. 50-60% pipeline occupation under pressure computing to/from registers and cache only, and corresponding speed compared to a conventional machine.

§4.3 **Branches** are an obstacle to hardware for encrypted computing and HEROIC implements a (signed) comparison relation $x^{\mathcal{E}}\,[\leq]\,y^{\mathcal{E}}$ via a lookup table for arithmetic sign (positive or negative) for branch decisions. It acts as an extra key. The table reports whether $x^{\mathcal{E}}$ has x positive or negative. In order that the table be small enough, HEROIC's encryption is one-to-one, not one-to-many. The table still contains 2^{15} rows of 2048 bits (256 bytes) each, so it is 8 MB. The architecture is a stack machine, not a von Neumann machine (the conventional architecture for modern computers). That has the advantage that local variables within a software function are accessed by a (plaintext) number in the machine code: the stack relative offset of the variable. In a conventional architecture, the location would be accessed via a computed memory address, which would be a 2048-bit encrypted number because the processor does its computation encrypted, so 2^{2048} memory locations would be addressed and only a few of those could physically exist.

Cryptoblaze passes branching decisions across the internet to the user for resolution (after decryption). The KPU resolves branch decisions internally in the pipeline, via the modified arithmetic.

§4.4 **Memory addressing** is done encrypted in HEROIC – it connects only the last 22 bits of an encrypted address to memory address lines, which proves sufficient in practice to disambiguate the only 2^{16} possible different 2048-bit addresses. That is 16MB of addressable memory, consisting of 2^{16} locations each 256B (2048 bits) wide. Those addresses are scattered randomly through 1GB of physical RAM, but HEROIC's address translation unit (TLB – translation lookaside buffer) remaps them to a contiguous, physically backed, 16MB subspace. The KPU remaps 128-bit encrypted addresses to a designated 32-bit region.

§4.5 **The TLB** always has to be a special design. Processors for encrypted computing have an addressing problem – it is impossible to provide physical backing for all the encrypted address range. Those addresses that do occur in a program must each be remapped individually to a backed region of memory. But conventional TLB technology remaps addresses 8192 (a 'page') at a time, so prototypes have to innovate.

A common solution is unit granularity in the TLB, plus dynamic remapping. Each encrypted address is mapped when it is encountered for the first time to the next free address in physically backed memory. Releasing defunct mappings in the TLB is one of the problems for these processors.

§4.6 **A special machine code instruction set** is needed. HEROIC's, comprises just the one form: $x{\leftarrow}x[-]y$ conditionally followed by a jump to a point elsewhere in the program if the result is not positive. That is computationally complete [34], and the aim is not only to simplify the hardware but to make all programs look alike. Unfortunately HEROIC's one-to-one encryption undoes that, while compilation for the unusual instruction set is a challenge. The newer CryptoBlaze processor adapts a more conventional architecture and instruction set but the latter likely does not have the security properties (Box II) that are now seen as necessary in encrypted computing in order to resist *chosen instruction attack* (CIA) [23], as explained in the next paragraph. The KPU adopts one feature of HEROIC's instruction set (input and output may be shifted by arbitrary amounts by varying the instruction constants) and modifies the OpenRISC (`http://openrisc.io`) instruction set in line with that in order to resist CIA, as explained below.

The right machine code instruction set is pivotal when the operator may be an adversary, as conventional instructions are not secure. The operator may, for example, observe an (encrypted) user datum $x^{\mathcal{E}}$ and put it through the machine's division instruction to get $1^{\mathcal{E}} = x^{E}\,[/]\,x^{\mathcal{E}}$. Then any desired encrypted y may be constructed by repeatedly applying the machine's addition instruction for $y^{\mathcal{E}} = 1^{\mathcal{E}}[+]\dots[+]1^{\mathcal{E}}$. By using the instruction set's comparator instructions (testing $\mathcal{E}[2^{31}]\,[\leq]\,z^{\mathcal{E}}$, $\mathcal{E}[2^{30}]\,[\leq]\,z^{\mathcal{E}}$, $\dots$) on an encrypted z and subtracting on branch, z may be obtained efficiently bitwise. That is the chosen instruction attack (CIA) of [23]. If there is no division operator in the hardware then there will be a library routine for it (the attacker can write one themself given an encrypted 1). Failing that, they can try every encrypted value in a program and its trace and in practice one of those will be $1^{\mathcal{E}}$ with probability greater than $1/2^{32}$, and that gets a $1^{\mathcal{E}}$ at frequency better than blind guessing.

The right instruction set resists such attacks. The KPU's instruction set contains HEROIC's as a subset and is proved to make those attacks impossible (Theorem 1 below).

§4.7 **The compiler** must be involved too in order to avoid *known plaintext attacks* (KPAs) [35] based either on the idea that not only do instructions like $x^{\mathcal{E}}\,[-]\,x^{\mathcal{E}}$ predictably favor one value over others (the result there is always $x^{\mathcal{E}}\,[-]\,x^{\mathcal{E}}{=}0^{\mathcal{E}}$), but human programmers favor values like 1. The compiler must even out the statistics.

The compiler must do so even for object code consisting of a single instruction. That gives necessary conditions on instruction design and execution shown in Box II [7]. These constraints must be implemented by the hardware: instructions must (IIa) execute atomically, or recent attacks such as Meltdown [36] and Spectre [37] against Intel become feasible (in those, memory access instructions in a speculatively executed branch when aborted leave behind a 'halfway-done' taint in the form of cache lines loaded), must (IIb) work with encrypted values or an adversary could read them, and (IIc) must be adjustable via embedded encrypted constants to offset the values beneath the encryption by arbitrary deltas. The condition (IId) is for the security proofs and amounts to different padding or blinding factors for encrypted program constants and runtime values.

In this document (IId) will be further strengthened to:

No collisions between constants in different instructions or different positions. (IId*)

'Different instruction' means different opcodes. Padding beneath the encryption enforces

Box II: Machine code axioms. Instructions ...

(a) *Are a black box from the perspective of the programming interface, with no intermediate states.*

(b) *Take encrypted inputs to encrypted outputs.*

(c) *Are adjustable via (encrypted) embedded constants to produce any desired offset delta in the (decrypted, plaintext) inputs and outputs at runtime.*

(d) *Can have no cipherspace collisions between encrypted instruction constants and runtime data.*

that, and the processor silently produces nonsense on violation. The aim is to block experiments with transplanted program constants. With (IId), moving runtime encrypted data into instructions or vice versa was already blocked.

Remark 1. HEROIC's $x \leftarrow x\,[-]\,y$ instruction fails (IIc) because $x\,[+]\,C \leftarrow (x\,[+]\,A)\,[-](y\,[+]\,B)$ as (IIc) requires cannot be achieved by varying the constants in the instruction, as there are none. That is $x \leftarrow x\,[-]\,y\,[+]\,K$ where $K = A\,[-]\,B\,[-]\,C$ so that would be 'fixed' for (IIc) if the instruction included an extra additive constant K. But the subsequent test $x\,[\leq]\,0$ also needs to be fixed to $x\,[\leq]\,A$ for A supplied in the instruction. With that, HEROIC's instruction set would work for the argument and theorem of [9] (below).

The effect of (IIa-IId) is proved (Appendix, [9]) to be:

Theorem 1. *A program and its runtime trace may consistently be interpreted arbitrarily in terms of data beneath the encryption at any one point in memory or trace.*

The technical argument shows that picking any point in the trace, so far as the adversary not privy to the encryption can tell, the word beneath the encryption may vary over a 32-bit range across recompilations, equiprobably.

'Chaotic' compilation always threatens the adversary that a delta offset has been introduced into runtime data beneath the encryption by varying the constants in the instruction before and after a point of interest, because (IIa) and (IIb) prevent the adversary knowing and (IIc) allows the variation (note (IIa) means 'no side-channels').

Theorem 2 (ρCSS)**.** *Relative cryptographic semantic security holds for any one word of data beneath the encryption and an adversary not privy to the encryption.*

That is what is usually rendered as *encrypted computation does not compromise encryption*, but it is really trivial. If one imagines the program that does nothing, consisting of no instructions, which transmits input to output unchanged, all it says is that the input can be any value (and the output will be the same any value). There is no reason or way for an adversary to discern any tendency beneath the encryption to some proper subset of values.

But data words in a program of any size and form are individually unconstrained by the adversary's observations (or experiments, as a continuation of the argument deduces) according to Theorem 1. Further, intuitively the adversary can select any two points in the program, except a pair as remarked, and they can be varied independently via changes in the surrounding instructions that the adversary cannot perceive because the instruction constants are encrypted, and this paper will quantify that intuition.

A 'chaotic' compiler backs the threat by really varying runtime data beneath the encryption independently and arbitrarily across recompilations to the extent the laws of programming allow, as ($\mathcal{S}$) ideates. How the compiler organizes that is encapsulated in Box III: a new *obfuscation scheme* is generated at each recompilation.

Definition 4 (Obfuscation scheme)**.** An *obfuscation scheme* is a plan that specifies *a delta from nominal for the data beneath the encryption in every memory and register location per point in the program control graph, before and after every instruction.*

Box III: What the compiler does, in sequence:

i. *Generate an obfuscation scheme of planned data offsets from nominal beneath the encryption.*
ii. *Vary instruction constants to implement (i), thereby leaving runtime traces unchanged in form, but not content.*
iii. *Equiprobably generate all variations (ii), hence schemes (i).*

A high-level, declarative, description of how a compiler works in this setting is that the compiler $\mathbb{C}[-]$ translates, for example, a source code expression e of type Expr, the value of which is to end up in register r at runtime, to machine code mc of type MC, and also generates a 32-bit offset Δe of (integer) type Off for r at that point in the program:

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

Let $s(r)$ be the content of register r in state s of the processor at runtime. The machine code mc's action is to change state s_0 to an s_1 with a ciphertext in r whose plaintext value differs by Δe from the nominal value $s_0(e)$ (the arrow symbol means 'steps eventually to'):

$$s_0 \overset{mc}{\rightsquigarrow} s_1 \text{ where } s_1(r) = \mathcal{E}[s_0(e) + \Delta e] \tag{2}$$

Remark 2. Bitwise exclusive-or or the binary operation of another mathematical group are alternatives for '+'.

For comparison, an 'ordinary', non-chaotic, compiler and ordinary execution platform would instead have the following abstract description:

$$\begin{aligned} \mathbb{C}[e]^r &= mc \\ s_0 \overset{mc}{\rightsquigarrow} s_1 &\text{ where } s_1(r) = s_0(e) \end{aligned}$$

The 'nominal value' $s_0(e)$ is formalized via a canonical construction. For the encrypted computing context map variable x to its register r_x (the runtime value is offset by a delta Δx), checking the (ciphertext) content of r_x in the state and discounting the delta from the plaintext value to get $s_0(x) = \mathcal{D}[s_0(r_x)] - \Delta x$. Arithmetic in the expression is formalized recursively, with $s_0(e_1 + e_2) = s_0(e_1) + s_0(e_2)$, etc. In the 'ordinary' context not encrypted computing, the nominal value of the variable is instead $s_0(x) = s_0(r_x)$ with no offset from the value in the register at runtime, and no encryption in the latter.

The encryption $\mathcal{E}$ is shared with the user and the processor but not the potential adversaries, the operator and operating system. The obfuscation scheme is known to the user, but not the processor and not the operator and operating system. The user compiles the program according to the scheme and sends it to the remote processor with the encrypted data to execute it on and needs to and does know the offsets at least on inputs and outputs. That allows the right data to be created and sent off for processing and allows sense to be made by the user of output received, once they have decrypted it.

5 Instruction Set

As noted in Section 4, conventional instruction sets are not safe against chosen instruction attacks (CIAs) in an encrypted computing setting. Without knowing any encrypted constants, it is still possible to program calculations that give a known constant as answer, such as $x^{\mathcal{E}}\,[-]\,x^{\mathcal{E}}$, or are biased stochastically towards a known subset. But instruction sets satisfying (IIa-IId) do not have that problem, by Theorem 1, so what is needed is

Table 1: Integer subset of a machine code instruction set for encrypted working.

op.	*fields*	*mnem.*	*semantics*
add	$\mathrm{r}_0\,\mathrm{r}_1\,\mathrm{r}_2\ k^{\mathcal{E}}$	add	$r_0 \leftarrow r_1\,[+]\,r_2\,[+]\,k^{\mathcal{E}}$
addi	$\mathrm{r}_0\,\mathrm{r}_1\ k^{\mathcal{E}}$	add imm	$r_0 \leftarrow r_1\,[+]\,k^{\mathcal{E}}$
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	$pc \leftarrow j$

LEGEND

r	– register indices
k	– 32-bit integers
pc	– prog. count reg.
j	– program count
'$\leftarrow$'	– assignment
ra	– return addr. reg.
$\mathcal{E}[\ ]$	– encryption
i	– pc increment
r	– register content
$k^{\mathcal{E}}$	– encrypted value $\mathcal{E}[k]$
$x^{\mathcal{E}}\,[o]\,y^{\mathcal{E}} = \mathcal{E}[x\,o\,y]$	
$x^{\mathcal{E}}\,[R]\,y^{\mathcal{E}} \Leftrightarrow x\,R\,y$	

a practical instruction set architecture (ISA) conforming to (IIa-IId). HEROIC's 'one instruction' instruction set can be modified to conform by the incorporation of a couple of encrypted constants in each instruction, as remarked in Remark 1, but it is untried and impractical as a compilation target.

A 'fused anything and add' general purpose ISA suitable for encrypted computing and satisfying conditions (IIa-IId) is put forward in [7] as a modification to OpenRISC v1.1 `http://openrisc.io/or1k.html`. A subset is shown in Table 1 and in all there are about 200 instructions, comprising single and double precision integer and floating point and vector subsets, uniformly 32 bits long. Instructions reference up to three of 32 general purpose registers (GPRs). There are just two instructions (load/store: **lw**/**sw**) for memory access. The instruction opcode is in the clear so the decoding unit in the processor pipeline can act on it, but that allows an adversary to see what kind of instruction it is, distinguishing addition from multiplication, etc. HEROIC's instruction set can be mimicked by emitting only instruction pairs **add** $\mathrm{r}_0\,\mathrm{r}_0\,\mathrm{r}_1\ k^{\mathcal{E}}$; **blei** $\mathrm{i}\,\mathrm{r}_0\ a^{\mathcal{E}}$ (the latter is the one-operand, one-constant form of the **ble** instruction).

To make this information concrete for the reader, a runtime trace for the Ackermann function[2] [38] compiled for this instruction set is shown in Table 2. The machine code is shown disassembled at left, register updates at right. Encrypted constants are shown with plaintext exposed and padding hidden. The constants in the instructions have randomized plaintexts, not 0s, 1s, 2s, etc. as would be expected. That goes for the updates too, except that for readability the delta in the obfuscation scheme for the return value in register **v0** is set to zero, and the (encrypted) '13' result can be seen. Ackermann's is the most computationally complex function possible, stepping up in complexity for each increment of the first argument, so getting the answer right is a confirmation of the correctness of the 'chaotic' compilation technique. It is short, but the code tests conditionals, assignments, arithmetic, comparators, call and return.

The 32-bit word-sized instructions may need to embed 128-bit or longer encrypted constants, so 'prefix' words are added as needed, carrying 29 extra bits of data each.

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

PC	instruction					update trace
...						
35	addi	t0	a0		$-86921031^{\mathcal{E}}$	t0 = $-86921028^{\mathcal{E}}$
36	sub	t1	t1	t1	$-327157853^{\mathcal{E}}$	t1 = $-327157853^{\mathcal{E}}$
37	beq	t0	t1	2	$240236822^{\mathcal{E}}$	
38	sub	t0	t0	t0	$-1242455113^{\mathcal{E}}$	t0 = $-1242455113^{\mathcal{E}}$
39	b 1					
41	sub	t1	t1	t1	$-1902505258^{\mathcal{E}}$	t1 = $-1902505258^{\mathcal{E}}$
42	xor	t0	t0	t1	$-1734761313^{\mathcal{E}}$	$1242455113^{\mathcal{E}}$ $1902505258^{\mathcal{E}}$ t0 = $-17347613130^{\mathcal{E}}$
43	beq	9	t0	t1	$167743945^{\mathcal{E}}$	
53	addi	sp	sp		$800875856^{\mathcal{E}}$	sp = $1687471183^{\mathcal{E}}$
54	addi	t0	a1		$-915514235^{\mathcal{E}}$	t0 = $-915514234^{\mathcal{E}}$
55	sub	t1	t1	t1	$-1175411995^{\mathcal{E}}$	t1 = $-1175411995^{\mathcal{E}}$
56	beq	2	t0	t1	$259897760^{\mathcal{E}}$	
57	sub	t0	t0	t0	$11161509^{\mathcal{E}}$	t0 = $11161509^{\mathcal{E}}$
...						
143	addi	v0	t0		$42611675^{\mathcal{E}}$	$\boxed{\texttt{v0} = 13^{\mathcal{E}}}$
...						
147	jr	ra				# return $13^{\mathcal{E}}$ in v0

LEGEND
(registers)
a0 – function argument
sp – stack pointer
t0, t1 – temporaries
v0 – return value (framed).

5.1 Instruction Diddling

Condition (IIb) of Box II requires one more constant in each branch instruction, an encrypted bit k_0 that decides if the 1-bit result of the test should be inverted. The test is observable by whether the branch is taken or not, so by (IIc) it should be modifiable by the compiler via an encrypted instruction constant. The extra bit changes equals to not-equals and vice versa, a less-than into a greater-than-or-equal-to, and so on. The bit *diddle*s the instruction. In practice, the bit is composed from the padding bits in the other constants in the instruction, so it is not explicit in Table 1, where the branch semantics shown are *post-diddle*, but the compiler knows what it is.

There is an argument that whether the first program code block after the branch instruction is the test 'fail' or 'succeed' case is already hidden by the general method of 'chaotic' compilation applied to boolean expressions. That argument is pursued below.

5.2 The Contestable Equals

Diddling works well to disguise less-than instructions and other order inequalities, but not well for equals versus not-equals. What the instruction is, equals or not-equals, may be tested by what proportion of operands cause a jump at runtime. If almost all do then that is a not-equals. If few do then that is an equals. Trying the same operand both sides is almost guaranteed to cause equals to fail because of the embedded constants k_1, k_2 in ($\doteq$), so if it succeeds instead, that (seeming!) equality test is (likely) diddled to not-equals.

So if the test succeeds or not at runtime is detectable in practice for an equals/not-equals branch instruction, contradicting (IIb). To beat that, a compiler must randomly change the way it interprets the original boolean source code expression at every level so it cannot be told if the source code, not the object code, had an equality or an not-equals test. It internally 'tosses a coin' as it works upwards through a boolean expression for if the source code at that point is to be interpreted by a truthteller or a liar. It equiprobably generates, at each level in the boolean expression, liar code and uses the branch-if-not-equal machine code instruction for an equality test in the source code, or truthteller code and uses the branch-if-equal instruction. The technique is a generalization of Yao's *garbled circuits* [39], but the compiler works with deeply structured and recursive logic as well as finite, flat, boolean normal forms of hardware logic gates.

With that strategy, if the equals branch instruction jumps or not at runtime does not relate statistically to what the source code says. Condition (IIc) of Box II on the output of the instruction is effectively vacuous with respect to the source code, as an observer who sees a jump take place does not know if that is the result of a truthteller's interpretation of an equals test in the source code and it has come out true at runtime, or it is the result of the liar's interpretation and it has come out false. Still, it might be preferable to remove equals/not-equals.

For other comparison tests, just as many operand pairs cause a branch one way as the other,[3] which makes it indistinguishable as to whether the opcode is diddled or not. An equality test cannot be recreated by an adversary as $x{\le}y$ and $y{\le}x$ because only $x{\le}y{+}k$ is available, for an unknown constant k. Reversed operands is allowed by (IId*) but produces $y{\le}x{+}k$, not $y{+}k{\le}x$. An estimate for k might ensue from the proportion of pairs (x, y) that satisfy the conjunction of the inequality and its reverse, and in particular $k{<}0$ would be signaled by the complete absence of pairs that simultaneously satisfy both. But diddling means the conjunctions might be $x{>}y{+}k$ and $y{>}x + k$ instead, and those have no solutions when $-k{-}1$ is negative, not when k is negative. So equiprobably $k{<}0$ or $k{\ge}0$, which gives nothing away.

Remark 3. A boolean 'liar' adds a delta equal to 1 mod 2 to 1-bit data beneath the encryption, a 'truthteller' adds 0 mod 2.

6 Chaotic Compilation

This section will describe in more detail but still declaratively and abstractly what a chaotic compiler for encrypted computing does, hopefully pointing out for a security audience just what is difficult and what is easy about it.

The point of note is that the compiler works with a 'deltas' database $D : \text{Loc} \to \text{Off}$ containing an integer offset Δl of integer type Off for data in register or memory location l (type Loc). The offset is the delta by which the runtime data plaintext beneath the encryption in the location is to vary from nominal at runtime, following the description in (1,2), and the database D is the incarnation of the *obfuscation scheme* of Defn. 4 for this point in the program code/control graph. The compiler has to remember the offset deltas as it works through the code, and this database serves as scratchpad.

Routinely, the compiler (any compiler) also maintains a 'location' database $L : \text{Var} \to \text{Loc}$ mapping source variables to register and memory locations. An intermediate layer in the compiler handles that and that matter is entirely elided here.

The reader uninterested in the detail that is provided can skip it. But the detail is required in order to prove what will be claimed, namely that the compilation method implements the tactic ($\mathcal{T}$). There is no way of doing that other than by giving detail.

6.1 Expressions

Filling in (1) in more detail, compiling an expression e to code mc_e that will get the result in register r at runtime means the compiler does a computation

$$(mc_e, \Delta r) = \mathbb{C}^L[D : e]^r \tag{3}$$

where mc is machine code (type MC), a sequence of machine code instructions, and Δr is the integer offset (type Off) from nominal beneath the encryption that the compiler intends for the result in r at runtime. Recall that L is the location database mapping source code variables to register and memory locations, and D is the database containing the 'obfuscation vector' at this point in the code, a list of planned delta offsets at runtime beneath the encryption per location. The question is whether the compiler has freedom of choice in choosing Δr. It might be that instructions are not available in the instruction

[3]In 2s complement arithmetic $x < y$ is the same as $x - y = z$ and $z < 0$ and exactly half of the range satisfies $z < 0$, half satisfies $z \le 0$.

set by which it could vary Δr arbitrarily and equiprobably across recompilations.

To translate $x{+}y$, where x and y are signed integer expressions, the compiler emits machine code mc_1 computing expression x in register r_1 with offset Δr_1, and emits machine code mc_2 computing expression y in register r_2 with offset Δr_2. By induction that is:

$$(mc_1, \Delta r_1) = \mathbb{C}^L[D : x]^{r_1} \tag{3^x}$$

$$(mc_2, \Delta r_2) = \mathbb{C}^L[D : y]^{r_2} \tag{3^y}$$

It decides a random offset Δr in r for the whole expression e, emitting the compound code

$$mc_e = mc_1;\ mc_2;\ \mathbf{add}\ r\ r_1\ r_2\ k^{\mathcal{E}}$$

where $\mathbf{add}\,r\,r_1\,r_2\,k^{\mathcal{E}}$ is the integer addition instruction, with semantics $r{\leftarrow}r_1[+]r_2[+]k^{\mathcal{E}}$, and $k = \Delta r - \Delta x - \Delta y$ has been designed so the sum is returned in r with offset Δr beneath the encryption. That is:

$$(mc_{x+y}, \Delta r) = \mathbb{C}^L[D : x{+}y]^r \tag{3^+}$$

implementing (3). The offset Δr is freely chosen. This construct introduces one 'arithmetic instruction that writes', the **add**, and one arbitrarily mutable offset for it, Δr. That implements the tactic ($\mathcal{T}$). There is nothing special about the '+' here that has been taken as an example, so (3) holds inductively of all pure expressions. The (trivial) base case is for a simple variable reference x as expression e, which takes a single trivial '+0' addition instruction to bring it out of the register r_x to which it is mapped by L and into r. The compiler may substitute an arbitrarily chosen '$+k$' instead of '+0', thus setting the offset Δr in r as it wills, satisfying (3).

The construction also gives ($\mathcal{T}$), as Δr for $x+y$ is free and exactly one new 'instruction that writes' to r (the $\mathbf{add}\,r\,r_1\,r_2\,k^{\mathcal{E}}$) is involved for it in mc_e and the constant in that is freely varied without restriction, which is sufficient to vary Δr freely by (IIc). By induction every 'instruction that writes' in mc_x and mc_y already is freely varied without restriction. The base case for a single instruction reference likewise involves one addition instruction with a freely variable constant. In conclusion, ($\mathcal{T}$) holds inductively of all pure expressions.

Literal constants (0-ary arithmetic operations) as expressions are implemented by a completely random choice of value by the compiler in register r. The database D merely records the offset from the nominal ('intended') value.

6.2 Statements

The compiler for statements s changes the database D of deltas at multiple locations. The abstract, high-level description of what it does in delivering code mc_s is:

$$D' : mc_s = \mathbb{C}^L[D : s] \tag{4}$$

A less formally complete exposition will be given than for expressions, to relieve the reader. It merely has to confirm that ($\mathcal{T}$) is satisfied. Consider an assignment statement $z{=}e$ (which statement will be called s) to a source code variable z, which the location database L binds in register $r{=}Lz$. By induction code mc_e for evaluating expression e in temporary register t_0 at runtime is emitted via the expression compiler as in (3) with t_0 for r:

$$(mc_e, \Delta t_0) = \mathbb{C}^L[D : e]^{t_0} \tag{3^e}$$

A short form add instruction with semantics $r \leftarrow t_0\,[+]\,k^{\mathcal{E}}$ is emitted to change offset Δt_0 to a new randomly chosen offset $\Delta' r$ in register r:

$$mc_s = mc_e;\ \mathbf{add}\ r\ t_0\ k^{\mathcal{E}}$$

where $k = \Delta' r - \Delta t_0$. That implements (4) for assignment:

$$D' : mc_{z=e} = \mathbb{C}^L[D : z{=}e] \tag{4^{ass}}$$

where the change in the database of offsets is at r, to $D'r = \Delta' r$. The new offset $\Delta' r$ is freely and randomly chosen by the compiler, supporting ($\mathcal{T}$), and *one* new 'arithmetic instruction that writes,' the **add**, is accompanied by *one* new random delta, supporting ($\mathcal{T}$) (by induction, ($\mathcal{T}$) is already true of the code implementing e).

The compilation of code constructs if, while, goto, sequence, is entirely standard and is left as an exercise for the determined reader. The model of proof above is followed to show ($\mathcal{T}$) holds. A codicil is that at the end of both branches of conditionals, at the beginning and end of loops, at source and target of gotos, the offset deltas in the database D must coincide for reasons of correctness of the computational semantics, which limits the variability that the compiler can achieve. Within that constraint ($\mathcal{T}$) is satisfied. The information theory is discussed in Section 7, but the actual code constructions are skipped here, being clear to 'one skilled in the art'.

6.3 Ramified Types

The problem with types is that there are so many of them, and the approach to representing them on an encrypted platform is not intrinsically obvious. A 64-bit integer could be represented by encrypting the 64-bit plaintext into a 128-bit ciphertext, for example (the platform we have used for prototyping is physically 128-bit). Or it could be broken into two 32-bit parts that are encrypted separately as two 128-bit ciphertexts.

Likewise, the way the compiler ought to vary the data is not intrinsically clear. Should a single 32-bit offset be applied simultaneously to both 32-bit parts of a 64-bit number? Should the additive carry be passed between the parts, or ignored?

The answers lie with principle ($\mathcal{T}$): every instruction that writes in the trace must vary to the fullest extent possible. With the instruction set shown in Table 1, the same offset when writing both 32-bit halves of a 64-bit number would mean that the second write instruction could not vary in a way distinct from the first, contradicting ($\mathcal{T}$). So the delta offsets for the two 32-bit halves of a 64-bit number must be separate and independent.

A similar consideration says that every entry in an array (and every 32-bit part of that entry) must have its own separate, independent, delta offset. The problem is that the compiler does not know which array entry will be accessed at runtime, so cannot in principle compile for any particular delta offset. A solution would be a single, shared 32-bit delta offset for every entry in the array (and every part of every entry), so the compiler could predict the offset to apply. But that runs foul of ($\mathcal{T}$). It also means that when one entry is changed, since by ($\mathcal{T}$) the write ought to freely create a new offset, all the other entries in the array ought to be brought into line for the new array-wide delta offset with a 'write storm', even though the source says they are not being written to. That might be useful from the point of view of disguising which entry is intended to be written to, but it is not computationally 'efficient' to have linear time complexity writes to an array. On the other hand, reads are constant time complexity, which is attractive (and what a programmer expects).

The situation is worse again for pointers, which could point anywhere (the compiler cannot generally predict). The argument applied above would say that therefore every part of every data structure must all, universally, share the same delta offset, which makes nonsense of variability. The only substantial variation in delta offsets would be in registers, which pointers cannot reference, and memory would get a single delta offset to be applied everywhere. Another approach is needed and the successful one is discussed below.

6.3.1 Long types

Firstly, the reasoning above concluded that to satisfy ($\mathcal{T}$), double length (64-bit) plaintext integers x ought to be treated as concatenated 32-bit integer 'halves' $x = x^H \cdot x^L$, the high and low 32 bits respectively. The encryption $x^{\mathcal{E}}$ of x occupies two registers or two memory locations, containing the encrypted values $\mathcal{E}[x^H]$, $\mathcal{E}[x^L]$ respectively. That is not only not obvious but also needs notation with which to express the corresponding operational semantics, or the page would fill with H and L superscripts.

Definition 5. Encryption of 64-bit integers x comprises encryptions of the 32-bit high and low bit halves separately:

$$x^{\mathcal{E}} = \mathcal{E}[x] = \mathcal{E}[x^H \cdot x^L] = \mathcal{E}[x^H] \cdot \mathcal{E}[x^L]$$

Instructions that operate on encrypted 64-bit types contain (encrypted) 64-bit constants to satisfy (IIc), in order that 64-bit delta offsets across the range can be achieved. But they may and will be 'added' as high+high, low+low separately, with no carry. A carry is prohibitively difficult to manage in the compiler and it is not necessary from the point of view of range, and it is justified by Remark 2 (any binary operation of a mathematical group is valid).

Definition 6. Let $-^2$ and $+^2$ be the two-by-two independent application of respectively 32-bit addition and 32-bit subtraction to the pairs of 32-bit plaintext integer high-bit and low-bit components of 64-bit integers, with similar notation for other operators. E.g.:

$$(u_1 \cdot l_1) +^2 (u_2 \cdot l_2) = (u_1 + u_2) \cdot (l_1 + l_2)$$

Definition 7. Let $\tilde{*}$, $\tilde{+}$ etc. denote multiplication, addition, etc. on 64-bit integers written as two 32-bit integers.

Then a suitable atomic encrypted multiplication operation for the instruction set working on encrypted 64-bit operands $x^{\mathcal{E}}$, $y^{\mathcal{E}}$ and satisfying (IIa-IId) gives the result:

$$\mathcal{E}[(x -^2 k_1) \,\tilde{*}\, (y -^2 k_2) +^2 k_0] \tag{$\tilde{*}$}$$

Here k_0, k_1, k_2 are 64-bit plaintext integer constants embedded encrypted (per Defn. 5) in the instruction as $k_i^{\mathcal{E}}$, $i = 0, 1, 2$. Putting it in terms of the effect on register contents, a suitable encrypted 64-bit multiplication instruction is:

$$r_0^H \cdot r_0^L \leftarrow (r_1^H \cdot r_1^L \,[-^2]\, k_1^{\mathcal{E}}) \,[\,\tilde{*}\,](r_2^H \cdot r_2^L \,[-^2]\, k_2^{\mathcal{E}}) \,[+^2]\, k_0^{\mathcal{E}}$$

For 64-bit operations the processor partitions the register set into pairs referenced by a single name. In those terms, the multiplication instruction semantics simplifies to:

$$r_0 \leftarrow (r_1 \,[-^2]\, k_1^{\mathcal{E}}) \,[\,\tilde{*}\,](r_2 \,[-^2]\, k_2^{\mathcal{E}}) \,[+^2]\, k_0^{\mathcal{E}}$$

In the instruction set, that is the primitive, atomic instruction

$$\mathbf{mull}\, r_0\, r_1\, r_2\, k_0^{\mathcal{E}}\, k_1^{\mathcal{E}}\, k_2^{\mathcal{E}}$$

following the general assembly format and nomenclature of Table 1. The **l** suffix means it works on 'long', i.e., 64-bit, integers. The other arithmetic instructions follow the same pattern, and compiled code for long integer expressions and statements on the encrypted computing platform follows exactly the form for 32-bit but with '**l**' instructions. Just as for 32-bit, exactly one new encrypted (64-bit) 'arithmetic instruction that writes' is issued per compiler construct, and through it, the 64-bit (i.e., 2×32-bit) delta offset in the target may be freely chosen by the compiler, supporting ($\mathcal{S}$) and ($\mathcal{T}$).

6.3.2 Short Types

Machine code instructions that work arithmetically on 'short' (16-bit) or 'char' (8-bit) or 'bool' (1-bit) integers are not needed to compile the C language at least, because short integers are immediately promoted to 32-bit for arithmetic. The compiler generates only internal accounting for such *casts*.

6.3.3 Arrays and Pointers

The natural way to bootstrap integers to arrays $\mathtt{A}$ of n integers is to imagine a set of integer variables $\mathtt{A}_0$, $\mathtt{A}_1$, ... one for each entry of the array. That allows the compiler to translate a lookup $\mathtt{A}[i]$ as if it were code

$$\textbf{int}\ \mathtt{t} = \mathtt{i};\ (\mathtt{t} = \mathtt{0})?\mathtt{A}_0{:}(\mathtt{t} = \mathtt{1})?\mathtt{A}_1{:}\dots$$

using a temporary variable $\mathtt{t}$ and the C ternary operator '_?_:_'. A write $\mathtt{A}[i]{=}x$ can be translated as if it were

$$\textbf{int}\ \mathtt{t} = \mathtt{i};\ \textbf{if}(\mathtt{t} = \mathtt{0})\ \mathtt{A}_0 = x\ \textbf{else if}(\mathtt{t} = \mathtt{1})\ \dots$$

The entries in the array get individual offsets from nominal $\Delta\mathtt{A}_0$, $\Delta\mathtt{A}_1$, ... in the obfuscation scheme maintained by the compiler. Amazingly, that is right according to the discussion with respect to what ($\mathcal{T}$) implies at the beginning of this section, yet it is non-obvious. One reason why it is non-obvious to a compiler expert is that both on read and write the scheme is linear time in the size n of the array (it can, however, easily be improved to log complexity) and array access 'should' be constant complexity. It is also apparently going to be impractical for pointers, where the expressions and statements above would have to be 2^{32} elements long, as where pointers land at runtime cannot be predicted.

Nevertheless, the extra complexity may be acceptable in this context – array lookup ought ideally to at least simulate looking at each entry (or many of them) in the array in order to disguise which is read, so it should not be dismissed on that basis. Multi-core machines may even be able to execute the component elements simultaneously.

The important technical points of the scheme above are that (a) the in-processor equality test ignores differences between different encryption aliases of the index and (b) an invariant set of encrypted addresses $\mathcal{E}[\&\mathtt{A}_0]$, $\mathcal{E}[\&\mathtt{A}_1]$, etc. are passed to memory, so lookup is always to the same place even though memory does not decrypt addresses (c.f. the discussion in Section 1). (Memory access can be said to be subject to *hardware aliasing* [40] in the encrypted computing context: i.e., different encryption aliases of an address access different memory locations, and (a) and (b) combined beat that.)

The scheme works for pointers $\mathtt{p}$ too, with lookup $\mathtt{*p}$ being compiled like this:

$$(\mathtt{p} = \&\mathtt{A}_0)?\mathtt{A}_0{:}(\mathtt{p} = \&\mathtt{A}_1)?\mathtt{A}_1{:}\dots$$

The pointer $\mathtt{p}$ must be known to be in $\mathtt{A}$, so we have modified C to declare pointers along with a global range that they point into:

$$\textbf{restrict}\ \mathtt{A}\ \textbf{int}\ {}^*\mathtt{p};$$

There is still a problem with the scheme in general, however, with respect to the principle ($\mathcal{T}$). That is that every access to the ith entry $\mathtt{A}[i]$ is via precisely the same encryption alias $\mathcal{E}[\&\mathtt{A}_i]$ as address, and though it beats the hardware aliasing effect, which memory location it accesses is visible, hence counts in itself as a 'write', yet it does not vary as it might. The processor has to repeat exactly the following calculation to get the address right again and again. An improvement can be made, but first the code involved has to be listed explicitly (the reader can take the listing in the next few lines for granted and skip).

Say `A` starts at the nth stack location, so `A`$[i]$ is the $n{+}i$th (assuming word sized entries). The plaintext address is $sp{+}(n{+}i)$, where sp is the address of the base of the stack. Read should normally be via this pattern of load word instruction:

$$\mathbf{lw}\ r\ \mathcal{E}[n{+}i](\mathbf{sp}).$$

That causes the processor to sum $(n+i)^{\mathcal{E}}\,[+]\,s^{\mathcal{E}}$ where $s^{\mathcal{E}}$ is the value in **sp**. It passes the result as effective address. But the value in s differs from its nominal value sp by a $\Delta\mathbf{sp}$ planned by the compiler, so the read instruction must be:

$$\mathbf{lw}\ r\ \mathcal{E}[(n{+}i){-}\Delta\mathbf{sp}](\mathbf{sp})$$

The same ciphertext $\mathcal{E}[(n{+}i){-}\Delta\mathbf{sp}]$ must be used as the constant at every read.

For write, the compiler emits the corresponding store:

$$\mathbf{sw}\ \mathcal{E}[(n{+}i){-}\Delta\mathbf{sp}](\mathbf{sp})\ r$$

That works around the 'hardware aliasing' effect in encrypted computing but does not support ($\mathcal{T}$) because the effective address could be varied, as explained below.

6.3.4 Varying Addresses

To satisfy ($\mathcal{T}$) the compiler should vary the address used at every write, by choosing a new encryption alias for $\mathcal{E}[\&\mathtt{A}_i]$ so a new memory location is written. Reads will be from there till the next write. That does satisfy ($\mathcal{S}$).

An array or variable on the heap instead of stack necessitates the heap base address register (**zer**) instead of **sp** in the load/store instructions, otherwise code is the same.

Unfortunately, using a new address all the time quickly fills the address mapping cache (the TLB) at runtime with addresses that will never be used again yet occupy translation slots. The solution is given below and again involves the compiler.

At the ends of loops (and after the then/else blocks of conditionals, at return from a function, at the label target of **goto**s, and wherever two distinct control paths join) the compiler issues instructions to restore the original address used by copying the data back to there from the address currently in use. The original delta offsets also have to be restored, but we will suppose that is done separately. Say the variable in question's location is nominally the nth on the stack. The resynchronization instruction sequence is

$$\mathbf{lw}\ r\ \kappa_1(\mathbf{sp});\ \mathbf{sw}\ \kappa_0(\mathbf{sp})\ r$$

where κ_0, κ_1 are the encryption aliases for $\mathcal{E}[n{-}\Delta\mathbf{sp}]$ in use at the beginning and end of the loop respectively. That reads from the one address then writes to the other.

The solution to the problem that varying the address used to access arrays and variables fills the TLB with mappings that will not be used again is that the compiler issues an instruction sequence to remove the mapping for the defunct address:

$$\mathbf{addi}\ r\ \mathbf{sp}\ \kappa_1;\ \mathbf{mtspr}\ \mathbf{UDTLBEIR}\ r;$$

The addition instruction reproduces exactly the processor pipeline calculation that forms the effective address, and the 'move to special purpose register' (**mtspr**) instruction sends it to the special 'user data TLB entry invalidate register' (**UDTLBEIR**), which affects the TLB.

The register needs an instruction to be executed in operator mode for access to succeed, so a system call is required, but there is no information leak because in user mode the same instruction would be used and it only carries the effective address, which is visible by which memory location is accessed. The compiler can save up these sequences till the end of a code block or the function body in order to keep defunct entries longer in the TLB (the advantage is that of bank robbers who shake their pursuers by swapping getaway cars in a busy car park instead of a quiet cul-de-sac).

Table 3: Trace for sieve showing hidden padding bits in data (right). Stack read and write instruction lines are in red, address base (register content, right) in violet and address displacement (instruction constant, left) in blue.

```
PC     instruction                       trace updates              | hidden
...
19300 addi t1 sp ℰ[-407791003]           t1 ← ℰ[-866593752|1532548040]    # write local array
19320 sw    ℰ[866593746](t1) t0          mem[ℰ[-6|-712377144]]            # a[7] at sp+40
                                           ← ℰ[-866593745|1800719299]

...
20884 addi t1 sp ℰ[-1763599776]          t1 ← ℰ[2072564771|-1935092797]  # write local variable
20904 sw    ℰ[-2072564772](t1) t0        mem[ℰ[-1|1518992593]]            # i at sp+45
                                           ← ℰ[2072564779|-1773201679]

...
22340 addi t1  sp  ℰ[-418452205]         t1 ← ℰ[-877254954|1532548040]
22360 bne  t0  t1  84
22384 addi t1  sp  ℰ[-407791003]         t1 ← ℰ[-866593752|1532548040]   # read local array
22404 lw   t0  ℰ[866593746](t1)          t0 ← ℰ[-866593745|1800719299]   # a[7] at sp+40
22424 addi t0  t0  ℰ[-1668656853]        t0 ← ℰ[1759716698|1081155516]
22444 b    540
22988 addi t1  zer ℰ[1759716697]         t1 ← ℰ[1759716697|1325372150]
23008 bne  t0  t1  44
...
23128 addi t0  sp  ℰ[-1763599776]        t0 ← ℰ[2072564771|-1935092797] # read local variable
23148 lw   t0  ℰ[-2072564772](t0)        t0 ← ℰ[2072564779|-1773201679] # i at sp+45
23168 addi t0  t0  ℰ[1723411350]         t0 ← ℰ[-498991167|-981581771]
23188 addi t0  t0  ℰ[-1862832992]        t0 ← ℰ[1933143137|-1629507929]
23208 addi v0  t0  ℰ[-1933143130]        v0 ← ℰ[7          |1680883739]   # return
...
23272 jr   ra
STOP
```

6.3.5 Structs and Unions

C 'structs' are records with fixed fields. They cause no problem for the compiler at all. It treats each field in a variable of struct type like a separate variable. That is, for a variable `x` of struct type with fields `.a` and `.b` the compiler invents variables, `x.a` and `x.b`.

For an array `A` with N entries that are structs, the compiler invents $2N$ variables $\mathtt{A}_i$`.a` and $\mathtt{A}_i$`.b`. Access to `A`[i]`.b` causes the compiler to emit code that tests only the `.b` addresses in the range dedicated to `A`, half of the total.

Unions have a surprise. The correct code for accessing a union member is exactly that for accessing a variable `x.b` sited at the start of the union `x`. But they also provide another indication that the correct way to treat arrays and other long types is via one delta offset per entry, not one delta offset shared for every entry, despite the compiler problems and inefficiencies it causes. The reason is that a union of an array with a struct (a common programming meme) would force all the struct's fields to the same (single, unique) delta offset as the array. That goes against the principle ($\mathcal{T}$).

6.4 Memory Example

Running a Sieve of Eratosthenes program[4] for primes shows up well how memory accesses are affected by encrypted address aliasing.

The final part of the trace is shown in Table 3 with two reads from elements on the stack shown in red. The address base (in register) and address displacement (a constant in the load/store word instruction) are shown in violet. The assignments to these stack locations are up-trace and do have the same address base and displacements as in the later reads. The plaintext addresses -6, -1 reflect the fact that the stack grows down from top of memory (-1), but it is the $\mathcal{E}$[-6|-712377144] (address -6, padding -712377144) and $\mathcal{E}$[-1|1518992593], the encryptions of the compound of address and hidden bits, that are

[4] Sieve C code: **int** S(**int** n) { **int** a[N]={[0. . . N-1]=1,}; **if** (n>N||n<3) **return** 0; **for** (**int** i=2; i<n; ++i) { **if** (!a[i]) **continue**; **for** (**int** j= 2*i; j<n; ++j) a[j]=0; }; **for** (**int** i=n-1; i>2; --i) if (a[i]) **return** i; **return** 0; } .

passed as the effective addresses to the memory unit.

7 Information Theory of Obfuscation

In this section the degree of independence of data beneath the encryption in a runtime trace for a program compiled satisfying ($\mathcal{S}$) and ($\mathcal{T}$) will be quantified, improving on the known ρCSS result in [9]. A *trace* T is the runtime sequence of writes to registers and memory locations. If a location is read for the first time without it having previously been written in the trace, then that is an *input*. There are no relevant differences in instruction order or kind (opcode) in this context B.

Trace T is a stochastic random variable, varying across recompilations of the same source code by a chaotic compiler. The compiler chooses obfuscation schemes as described in previous sections, and the probability distribution for T depends on the distribution of those choices. After an assignment to a register r, the trace is longer by one: $T' = T^\frown\langle r{=}v^{\mathcal{E}}\rangle$. Let $\mathrm{H}(T)$ be the *entropy* of *trace* T in this setting. I.e., let f_T be the probability distribution of T, the entropy is the expectation

$$\mathrm{H}(T) = \mathbb{E}[-\log_2 f_T] \tag{5}$$

The increase in entropy from T to T' (it cannot decrease as T lengthens) is informally the number of bits of unpredictable information added. Only these fragments of information theory will be required:

Fact 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.*

Fact 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.*

If the offset Δr beneath the encryption is chosen randomly and independently with flat distribution by the compiler, so it has maximal entropy, then $\mathrm{H}(T') = \mathrm{H}(T) + 32$, because there are 32 bits of unpredictable information added via the 32-bit delta to the 32-bit value beneath the encryption, so the 32-bit sum of value plus delta varies with (32-bit) maximal entropy.

Although per instruction the compiler has free choice in accord with ($\mathcal{T}$), not all the register/memory write instructions issued by the compiler are jointly free as to the offset delta for the target location – it is constrained to be equal at the beginning and end of a loop, and in general at any point where two control paths join (F):

Definition 8. An instruction emitted by the compiler that adjusts the offset in location l to a final value common with that in a joining control path is a *trailer* instruction.

Trailer instructions come in *sets* for each location l at a control path join, with one member on each path. Each is last to write to l before the join. In particular, there are trailer instructions before return from a subroutine.

Because running through the same instruction or a different instruction with the same delta offset for the target location a second time does not add any new entropy (the delta is determined by the first encounter), the total entropy in a trace can be counted as follows:

Lemma 1. *The entropy of a trace compiled according to ($\mathcal{T}$) is $32(n + m)$ bits, where n is the number of distinct arithmetic instructions that write in the trace, counted once only per set if they are one of a set of trailer instructions for the same location, and once each if they are not, and m is the number of input words.*

Recall 'input' is provided by those instructions that read first in the trace from a location not written earlier in it.

Observing data at any point in the trace sees variation across recompilations. The principle ($\mathcal{T}$) asserts that every opportunity provided by an arithmetic instruction that writes is taken by the compiler to introduce new variation. At 'trailers' the compiler organizes several instructions to synchronize final deltas in different paths but that is sometimes unnecessary because a location will be rewritten before it is ever read again. In such cases, the variation the compiler introduces is not maximal because it could be increased by varying deltas independently. So consider that compiler constructions might be embedded in a context that reads all locations. Then trailer synchronization is necessary and the compiler introduces the maximal entropy possible:

Proposition 1. *The trace entropy of context-free compiler constructions that conform to ($\mathcal{T}$) is maximal with respect to varying the constants in the machine code.*

The proposition implies at least 32 bits of entropy in the variation beneath the encryption must exist in any location l where (1) the location has been written, or (2) read without a prior write. In (1) the datum is written by an instruction and the compiler generates variations in the obfuscating delta $D'l$ in the obfuscation scheme D' after the instruction, or is copied exactly from somewhere else that the compiler influences in that way. In (2) it is an input, which is subject to planned variations D_0l that must be satisfied by the provider of the input.

The following is obtained by structural induction in [7]:

Corollary 1. ($\mathcal{S}$) *The probability across different compilations by a compiler that follows principle ($\mathcal{T}$) 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 formally implies Theorem 2, relative to the security of the encryption. But a stronger result can now be obtained from the lemma and proposition above:

Definition 9. Two data observations in the trace are (delta) *dependent* if they are of the same register at the same point, input and output of a copy instruction, or the same register after the last write to it in a control path before a join and before the next write.

The variation in the trace observed at two (or n) independent points is maximal possible:

Theorem 3. *The probability across different compilations by a compiler that follows principle ($\mathcal{T}$) that any n particular 32-bit values in the trace have encryptions $\mathcal{E}[x_i]$, provided they are pairwise independent, is $1/2^{32n}$.*

Each dependent pair reduces the entropy by 32 bits.

8 Discussion

Theorem 3 quantifies exactly the correlation that exists in data beneath the encryption in a trace where the compiler follows the principle ($\mathcal{T}$) (every arithmetic instruction that writes is varied to the maximal extent possible across recompilations). It names the points in the trace where the compiler's variations are weak and statistical influences from the original source code may show through. For example, if the code runs a loop summing the same value again and again into an accumulator, then looking at the accumulator shows an observer $\mathcal{E}[a + ib + \delta]$ for a constant offset δ and increasing i. That is an arithmetic series with unknown starting point $a + \delta$ and constant step b and it is likely to be one of the relatively few short-stepping paths, with small b. That knowledge can be leveraged into a stochastically based attack on the encryption. But if the encryption has no weakness to that vector then there is no danger. Such a characteristic of the encryption would be expressed as 'there is no polynomial time method that determines a or b from a sequence $\mathcal{E}[a+ib]$ with probability significantly greater than chance' (as block size $n \to \infty$).

A compiler following the principle ($\mathcal{T}$) does as well as any may to avoid weaknesses based on relations such as the above between data at different points in the runtime program trace. The only way to eliminate them completely is to have no loops or branches in the object code, by Theorem 3. That would be a finite-length calculation or unrolled bounded loop with branches bundled as $\mathcal{E}[tx + (1-t)y]$ into the calculation, where $x^{\mathcal{E}}$ and $y^{\mathcal{E}}$ are the outcomes from the two branches executed separately, and t is the boolean test result. (Those are essentially the calculations available via FHEs.)

That speaks to classical concepts of obfuscation and security via the following argument, which shows that the operator does not win game G2 of Section 3.

Claim. *Encrypted computing as described in this paper is resistant to polynomial time (in the block/word size n) attacks by the operator on the runtime data beneath the encryption, provided the encryption itself is resistant to such attacks.*

Proof sketch. Suppose the adversarial operator has a polynomial time (in the number of bits n in a word on the platform) method of working out what the data beneath the encryption is in register r at some identified point in the trace of program P. That point of interest may even move (polynomially) with n, being specified perhaps as 'the point of last change in r in the first n^3 steps'. The operator knows program P and may have suggested it themself, before the user compiles it. The operator can see the compiled maximal entropy code $\mathbb{C}[P]$, and see it running and probe it by running it at will.

The user readies a sequence of compilations $\mathbb{C}[P_n]$ of P, as described in this paper, with the nth being for a n-bit platform as target, and P having been partially or completely unrolled as P_n with no loops or branches in the first e^n (i.e., super-polynomially many) machine code instructions. If the program predictably ends before then, it is to be unrolled completely. These are compilations of the same program P all with the same end-to-end semantics that could be produced entirely automatically by a monolithic compiler incorporating the unrolling and branch bundling in its front end. The operator is invited to apply their method and predict the values beneath the encryption at the chosen points in the runtime trace(s) of these compiled codes, which may differ only in consequence of the number n of bits in a word on the platform (i.e., the theory of arithmetic mod 2^n).

Theorem 3 implies the operator's method cannot exist as follows. There are no loops or branches (hence no delta dependencies, per the terminology of the theorem and Defn. 9) 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. So there is no reliable relation between the data and the operator cannot use it as a decryption aid. The operator is reduced to attacking the encryption with no further information. The encryption on its own is hypothesized to be resistant to polynomial time attacks, and the claim is proved by contradiction. □

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, but with the data on every pin and wire converted to encrypted form. It is known from the theory of Yao's garbled circuits [39] that the intended values cannot be deciphered without the garbling scheme, which equates to an obfuscation scheme of deltas here: '+1 mod 2' is boolean negation on a 1-bit boolean value, while '+0 mod 2' leaves the 1-bit boolean value unchanged. The obfuscation scheme is known only to the user, not the operator.

The argument could have been made in 1986 (the year of publication of [39]), if the hardware and electronic engineers had moved their idea on into general computation.

A subtlety is that encryption appears not to be necessary for the argument if one substitutes for ‘data beneath the encryption’ with ‘value as really intended by the user’. Yao’s garbled circuits are already garbled without any encryption and equally the code produced by the chaotic compiler is ‘obfuscated’, with an obfuscation scheme determining randomly chosen offsets from nominal at every point in the trace. It is a matter of conjecture for an observer as to what the user/programmer really meant when 2 is in the trace, because if the obfuscation scheme has -1 as delta at that point, then 3 was really meant by the user, and if -12 is the delta, then 14 was really meant. All options are equally feasible, and the compiler has rendered them equally probable. The practical problem is that the mode of compilation – with exponential unrolling – in the argument for the Claim of this section leads to unfeasibly long machine code programs, and in any case there is no guarantee that any particular word size n is not vulnerable, only that in the limit any particular attack method fails. But that may be moot too as hardware platforms are not arbitrarily extensible in word length (perhaps they may become so). Simulating a platform with word length n on a 32 bit platform does not satisfy the axiom of atomicity in Box II.

As a codicil, the treatment of short integer types here (they are promoted to standard integer length) prompts the question of whether entropy could be increased by changing to 64-bit or 128-bit plaintext words beneath the encryption, instead of 32-bit, and correspondingly sized delta offsets from nominal. That logic appears correct. The 32-bit range of variation of standard-sized integers would be swamped by a 64-bit delta introduced by the compiler and the looped stepping example $\mathcal{E}[a + ib + \delta]$ at the beginning of this section would have a 64-bit δ, so would have 2^{64} possible origin points for the path for any hypothetical step b, not 2^{32}.

Implementation

At the current stage of development, our own prototype compiler (see `http://sf.net/p/obfusc`) has near total coverage of ANSI C with GNU extensions, including statements-as-expressions and expressions-as-statements. It lacks **longjmp**, and computed **goto**. It is being debugged via the venerable *gcc* ‘c-torture’ test-suite v2.3.3 (`http://ftp.nluug.nl/languages/gcc/old-releases/gcc-2/gcc-2.3.3-testsuite.tar.bz2`), and we are presently about one quarter way through that.

9 Conclusion

How to compile all of ANSI C for encrypted computing without decryption on the memory address (or data) path has been set out here. That opens up the field for software development using the canonical toolchain of compiler, assembler, linker, loader with operating system support. It allows processors for encrypted computing to access memory at normal speeds. The only hardware support needed is a unit granularity address translation lookaside buffer that remaps encrypted addresses first-come, first-served to physically backed memory. The compiler inserts instructions that release the mappings.

The technical difficulty is that encrypted addresses vary for the same plaintext address and distinct calculations for the same address produce a different ciphertext variant. A systematic coding discipline is followed that overcomes that. The downside of the compiler solution is that array and pointer access are linear or log time in the array size, not constant time, but the upside is that all modes of access, e.g., via pointer or displacement in array, are as compatible as the programmer expects them to be, to any structural depth.

Further, the compiler has to randomly vary the code it generates as much as possible in order to provide security guarantees. A single principle for the compiler to follow has been enunciated – any arithmetic instruction that writes must be varied by the compiler to the maximal extent possible. It has been shown that then the compiler is 'best possible' in terms of introducing maximal possible entropy across recompilations to the data beneath the encryption in a runtime trace, and that swamps biases introduced by human programmers or other agencies. The theory quantifies exactly an existing 'cryptographic semantic security relative to the security of the encryption' result for encrypted computing, and implies that the adversarial operator or operating system cannot guess what user data is beneath the encryption, to any degree better than chance. That is perhaps so even in the absence of encryption, thanks to an 'obfuscation scheme' the compiler modifies the user's data with, which an adversary has no basis for distinguishing from the user's intention.

References

[1] C. W. Fletcher, M. van Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proc. 7th ACM Work. Scal. Trust. Comp. (STC'12).* New York: ACM, 2012, pp. 3–8.

[2] 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, no. 7781. Heidelberg/Berlin: Springer, Feb. 2013, pp. 123–138.

[3] N. G. Tsoutsos and M. Maniatakos, "Investigating the application of one instruction set computing for encrypted data computation," in *Proc. Int. Conf. Sec., Priv. Appl. Crypto. Eng.* Springer, 2013, pp. 21–37.

[4] ——, "The HEROIC framework: Encrypted computation without shared keys," *IEEE Trans. CAD IC Sys.*, vol. 34, no. 6, pp. 875–888, 2015.

[5] P. Breuer, J. Bowen, E. Palomar, and Z. Liu, "A practical encrypted microprocessor," in *Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16)*, C. Callegari *et al.*, Eds., vol. 4. SCITEPRESS, Jul. 2016, pp. 239–250.

[6] ——, "The secret processor will go to the ball: Benchmark insider-proof encrypted computing," in *Proc. 3rd Euro. Symp. Sec. Priv. & Affil. Works. (EuroS&PW'18).* CA, USA: IEEE Comp. Soc., Apr. 2018, pp. 145–152.

[7] ——, "On obfuscating compilation for encrypted computing," in *Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17)*, P. Samarati, M. S. Obaidat, and E. Cabello, Eds., INSTICC. Port.: SCITEPRESS, Jul. 2017, pp. 247–254.

[8] ISO/IEC, "Programming languages – C," International Organization for Standardization, 9899:201x Tech. Report n1570, Aug. 2011, JTC 1, SC 22, WG 14.

[9] 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., no. 11149. Cham, Ger.: Springer, Oct. 2018, pp. 192–211.

[10] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[11] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *Proc. 8th Int. Conf. Availability, Rel. and Sec. (ARES'13).* IEEE, Sep. 2013, pp. 390–397.

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

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

[14] R. Hartman, "System for seamless processing of encrypted and non-encrypted data and instructions," Jun. 29 1993, US Patent 5,224,166.

[15] M. Hashimoto, K. Teramoto, T. Saito, K. Shirakawa, and K. Fujimoto, "Tamper resistant microprocessor," 2001, US Patent 2001/0018736.

[16] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU-based attestation and sealing," in *2nd Int. Work. Hard. Arch. Sup. Sec. Priv. (HASP'13)*, 2013.

[17] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA*, vol. 11, 2013.

[18] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *HASP@ ISCA*, vol. 10, no. 1, 2013.

[19] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. ACM Conf. Comp. and Commun. Sec. (CCS'12).* New York: ACM, 2012, pp. 305–316.

[20] K. Kissell, "Method and apparatus for disassociating power consumed within a processing system with instructions it is executing," Mar. 9 2006, US Patent App. 11/257,381.

[21] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1108970.1108971

[22] 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. 365–377.

[23] S. Rass and P. Schartner, "On the security of a universal cryptocomputer: The chosen instruction attack," *IEEE Access*, vol. 4, pp. 7874–82, 2016.

[24] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography.* CRC press, 1996, chapter 10, section 2.2.

[25] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. 29th Ann. Int. Conf. Th. Appl. Crypto. Tech. (EUROCRYPT'10).* Springer, 2010, pp. 24–43.

[26] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978.

[27] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Ann. ACM Symp. Th. Comp. (STOC'09)*, NY, 2009, pp. 169–178.

[28] 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).* CA, USA: IEEE Comp. Soc., Aug. 2018.

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

[30] R. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.

[31] 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. Heidelberg/Berlin: Springer, 1999, pp. 223–238.

[32] 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).* IEEE, 2018, pp. 702–708.

[33] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *Proc. 22nd Ann. ACM Symp. Th. Comp.*, ACM. ACM, 1990, pp. 514–523.

[34] J. H. Conway, "Fractran: A simple universal programming language for arithmetic," in *Open Problems in Commun. and Comp.*, T. M. Cover and B. Gopinath, Eds. Springer, 1987, pp. 4–26.

[35] A. Biryukov, "Known plaintext attack," in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Boston, MA: Springer, 2011, pp. 704–705.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[37] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[38] Y. Sundblad, "The Ackermann function. a theoretical, computational, and formula manipulative study," *BIT Num. Math.*, vol. 11, no. 1, pp. 107–119, Mar. 1971.

[39] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Ann. Symp. Found. Comp. Sci.* IEEE, 1986, pp. 162–167.

[40] M. Barr, "Memory," in *Programming Embedded Systems in C and C++*, 1st ed., A. Oram, Ed. Sebastopol, CA: O'Reilly & Associates, Inc., 1998, ch. 6, pp. 64–92.

[41] W. J. Cody, "Analysis of proposals for the floating-point standard," *Computer*, no. 3, pp. 63–68, 1981.

[42] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[43] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," Cryptology ePrint Archive, Report 2016/421, 2016, https://eprint.iacr.org/2016/421.

[44] A. Costache, N. Smart, S. Vivek, and A. Waller, "Fixed point arithmetic in she scheme," Cryptology ePrint Archive, Report 2016/250, 2016, https://eprint.iacr.org/2016/250.

[45] S. Arita and S. Nakasato, "Fully homomorphic encryption for point numbers," in *Proc. Int. Conf. Info. Sec. Crypto. (InSCrypt'16)*, ser. LNCS, K. Chen, D. Lin, and M. Yung, Eds., vol. 10143. Cham, Ger.: Springer, 2017.

Appendix

This appendix contains proofs from work that cannot be referred to without breaking the anonymity rules, and/or that it seems better to avoid interrupting the text with. Accordingly, the referees should perhaps regard it as additional material that may be incorporated into a final text or referenced from there, as they may require.

A Proofs

Proposition A0. *No method of observation exists by which the operator (who does not possess the key) may decrypt program output from the 'fixed' HEROIC instruction set of Remark 1.*

Proof. (Sketch) Suppose program C is written using only the instructions addition of a constant $y{\leftarrow}x[+]k^{\mathcal{E}}$ and branches based on comparison with a constant $x[<]K^{\mathcal{E}}$ (a 'fixed' HEROIC set), which satisfy (IIa-IId) of Box II. The hypothetical method takes as inputs the trace T and code C. But a modified code C^* is constructed below such that (i) it has a trace T^* that 'looks the same' as T to the operator, modulo encrypted data, and (ii) the new code C^* 'looks the same' as C, modulo embedded encrypted constants, so the operator's method must give the same result applied to C^* and T^* as it does applied to C and T, which is $y^{\mathcal{E}}$, say. But the code C^* gives the output $\mathcal{E}[y{+}7]$ when run, not $y^{\mathcal{E}}$. So the method does not work.

The program C^* differs from C only in the encrypted constants $K^{\mathcal{E}}$ in the branch instructions. Otherwise it is the same as C. The constants K need changing to match x taking a value that is 7 more than before beneath the encryption. Changing K to $K^* = K{+}7$ achieves that. Branches jump (or not) as they did before the increase of the plaintext data everywhere by 7. The addition instructions are consistent as they are with the plaintext increase in both input and output. So the code does the same at runtime as C does, but on data that is everywhere $x[+]7^{\mathcal{E}}$ instead of x, as required for the contradiction. □

Corollary A0. *There is no method by which the privileged operator can alter program C using just add and compare with constant instructions to get output $y^{\mathcal{E}}$ for known y.*

Proof. Suppose for contradiction that the operator builds program $C^*{=}f(C)$ that returns $y^{\mathcal{E}}$. Then its constants $k^{\mathcal{E}}$ and $K^{\mathcal{E}}$ are found in C, because f has no way of arithmetically combining them (the no collisions condition (IId) means they cannot be combined arithmetically in the processor and the operator does not have the encryption key). Proposition A0 says the operator cannot read y from the output of C^*, yet knows what it is. Done by contradiction. □

Theorem A1. *There is no method by which the privileged operator can read plaintext runtime data from a program C built from instructions satisfying (IIa-IId), nor deliberately alter it to give an intended output $y^{\mathcal{E}}$ with y known.*

Proof. (Sketch) A modified code C^* is constructed that looks the same modulo encrypted constants, and has runtime trace T^* that looks the same as the original T modulo encrypted data. The argument goes as for Proposition A0 and Corollary A0.

In program C, every arithmetic instruction of the form

$$r_0{\leftarrow}(r_1[-]k_1^{\mathcal{E}})\,[\Theta](r_2\,[-]\,k_2^{\mathcal{E}})\,[+]\,k_0^{\mathcal{E}}$$

for operator Θ can be changed for C^* via adjustments in its embedded constants to accommodate every data value passing through registers and memory to be +7 more beneath the encryption than it used to be, as in the proof of Theorem A0 and Corollary A0. The change is from k_i to $k_i^*{=}k_i + 7$, $i = 0, 1, 2$.

A branch instruction in C with test $(r_1\,[-]\,k_1^{\mathcal{E}})\,[R](r_2\,[-]\,k_2^{\mathcal{E}})$ for relation R, the instruction is changed for C^* to $(r_1\,[-]\,\mathcal{E}[k_1^*])\,[R](r_2\,[-]\,\mathcal{E}[k_2^*])$ with $k_i^* = k_i + 7$, $i = 1, 2$, and the branch goes the same way at runtime in trace T^* for C^* as it did originally in trace T for C. Unconditional jumps are unaltered.

The outcome is a trace T^* that is the same as T modulo the encrypted data values, which by hypothesis cannot be read by the adversary (they differ by 7 from the originals, beneath the encryption). Code C^* looks the same too, apart from the embedded (encrypted) constants, which also cannot be read by the adversary. As in the earlier proof, a method $f(C,T)$ for decryption must give the same result as $f(C^*,T^*)$, yet the answers (the decrypted data) are different by 7 in the two cases, so method f cannot exist. $\square$

Lemma A1. *There is a compile strategy for machine code instruction sets satisfying (IIa-IId) such that the probability across different compilations that any particular 32-bit value x has its encryption $x^{\mathcal{E}}$ in a given register or memory location at any given point in the program at runtime is uniformly $1/2^{32}$.*

Proof. Consider the arithmetic instruction I in the program. Suppose that by modifying the embedded constants in the other instructions in the program it is already possible for all other locations l other than that written by I and at all other points in the program to vary the value $x_l = x+\Delta x$, where $x_l^{\mathcal{E}}$ is stored in l, randomly and uniformly across compilations, taking advantage of the properties of the instruction set as the compiler described in the text does. Let I write value $y^{\mathcal{E}}$ in location l. By design, I has a parameter $k^{\mathcal{E}}$ that may be tweaked to offset y from the nominal result $f(x+\Delta x)$ by any chosen amount Δy. The compiler chooses k with a distribution such that Δy is uniformly distributed across the possible range. The instructions in the program that receive $y^{\mathcal{E}}$ from I may be adjusted to compensate for the Δy change by changes in their controlling parameters. Then $p(y{=}Y){=}p(f(x{+}\Delta x){+}\Delta y{=}Y)$ and the latter probability is $p(y{=}Y){=}\sum\limits_{\Upsilon} p(f(x{+}\Delta x){=}\Upsilon \wedge \Delta y{=}Y{-}\Upsilon)$. The probabilities are independent (because Δy is newly introduced just now), so that sum is $p(y{=}Y){=}\sum\limits_{\Upsilon} p(f(x{+}\Delta x){=}\Upsilon)p(\Delta y{=}Y{-}\Upsilon)$. That is $p(y{=}Y){=}\frac{1}{2^{32}}\sum\limits_{\Upsilon} p(f(x{+}\mathrm{d}x){=}\Upsilon)$. Since the sum is over all possible Υ, the total of the summed probabilities is 1, and $p(y{=}Y){=}1/2^{32}$. The distribution of $\mathcal{E}[x_l] = \mathcal{E}[x{+}\Delta x]$ in other locations l is unchanged. At a point where two control paths join the choice of Δy is not free, but instead must coincide in the second path to be compiled with the choice already made by the compiler in the first path to be compiled, which was, however, free. If the first path does not write l at all then let an 'add zero' instruction be inserted in it. Done by structural induction on the machine code program. $\square$

That compile strategy proves Theorem 1 of the text, and also:

Theorem A2 (**2** of the main text)**.** *Runtime user data beneath the encryption is* semantically secure *against the operator for programs compiled by the chaotic compiler of Lemma A1.*

Proof. Consider a probabilistic method f that guesses for a particular runtime value beneath the encryption 'the top bit b is 1, not 0', with probability $p_{C,T}$ for program C with trace T. The probability that f is right is

$$\mathrm{p}((b_{C,T}{=}1 \text{ and } f(C,T){=}1) \text{ or } (b_{C,T}{=}0 \text{ and } f(C,T){=}0))$$

Splitting the conjunctions, that is

$$\begin{aligned} &\mathrm{p}(b_{C,T}{=}1)\mathrm{p}(f(C,T){=}1 \mid b_{C,T}{=}1) \\ +\, &\mathrm{p}(b_{C,T}{=}0)\mathrm{p}(f(C,T){=}0 \mid b_{C,T}{=}0) \end{aligned}$$

But the method f cannot distinguish the compilations it is looking at as the codes and traces are the same, modulo the (encrypted) values in them, which the adversary cannot read. The method f applied to C and T has nothing to cause it to give different answers but incidental features of encrypted numbers and its internal spins of a coin. Those are *independent* of if the bit b is 1 or 0 beneath the encryption, supposing the encryption is effective. So

$$\begin{aligned} \mathrm{p}(f(C,T) = 1 \mid b_{C,T} = 1) &= \mathrm{p}(f(C,T) = 1) = p_{C,T} \\ \mathrm{p}(f(C,T) = 0 \mid b_{C,T} = 0) &= \mathrm{p}(f(C,T) = 0) = 1 - p_{C,T} \end{aligned}$$

By Lemma A1, 1 and 0 are equally likely across all possible compilations C, so the probability f is right reduces to

$$\tfrac{1}{2}\, p_{C,T} + \tfrac{1}{2}\,(1 - p_{C,T}) = \tfrac{1}{2}$$

since $\mathrm{p}(b_{C,T}{=}1) = \mathrm{p}(b_{C,T}{=}0) = \frac{1}{2}$. $\square$

The information theory in the text is based on the idea that instructions are varied by the compiler by changing the (encrypted) constants embedded in them, which additively varies the difference from 'nominal' of the result of the instruction through the full possible (32-bit) range (IIb). The viewpoint is that of an observer who can see the plaintext values beneath the encryption, because an encrypted word depends one-to-one on the plaintext when padding is taken into account.

Lemma A2 (Lemma 1 of text)**.** *The entropy of a trace is that from the instructions that appear for a first time in it.*

Proof. The delta Δs from the nominal state s in the experienced state $s{+}\Delta s$ is what contributes to entropy, because the nominal values s themselves are determined. The delta is introduced by an instruction f that nominally has semantics $s_0 \overset{f}{\mapsto} s_1$ but has been varied by the compiler to semantics f' such that $s_0{+}\Delta s_0 \overset{f'}{\mapsto} s_1{+}\Delta s_1$ with $s_1 = f(s_0)$. The compiler arranges the perturbation Δf in the constants of the instruction so that

$$\Delta s_1 = D' = f'(s_0) - f(s_0)$$

where D' is the obfuscation scheme at the point in the program just after the instruction, and $\Delta s_0 = D$ is that just before it. Both D, D' are independent of s_0, s_1. The state change experienced

$$(s_1 + \Delta s_1) - (s_0 + \Delta s_0)$$

is the part of the trace due to the instruction and, substituting, it is

$$f(s_0) - s_0 + D' - \Delta s_0$$

The Δs_0 is produced by previous instructions and $f(s_0) - s_0$ is the 'nominal' trace from an unperturbed instruction semantics f, leaving D' as the source of the entropy contribution by this instruction. At a second appearance in the trace of the instruction the variation D' is the same, and could have been predicted, so contributes no entropy.

For an instruction that reads input $x + \Delta x$ from memory location l for the first time, to register r, say, the data is offset by $\Delta x = Dl$ from the nominal value x, where D is the compiler's obfuscation scheme at the point in the program just before the instruction runs, and the same argument applies. □

B Floating Point

A practical instruction set for encrypted computing must also manipulate floating point numbers and this is how it works. Let single precision floating point instructions **addf**, **subf**, **mulf** etc. be denoted by a trailing **f**. They work on registers containing (encrypted) 32-bit integers that encode single precision floating point numbers ('float') per the IEEE 754 standard [41, 42].

Definition B1. Let $\dot{*}$, $\dot{+}$ etc. denote the floating point operations on plaintext integers encoding IEEE 754 floats.

Let $[\dot{*}]$, $[\dot{+}]$ etc. be the corresponding operations in the ciphertext domain, in the convention noted at end of Section 2.0.0.9. The multiplication **mulf** $r_0\, r_1\, r_2\, k_1^{\mathcal{E}}\, k_2^{\mathcal{E}}\, k_0^{\mathcal{E}}$ has semantics conforming to (IIa-IId) as follows:

$$r_0 \leftarrow (r_1\,[-]\,k_1^{\mathcal{E}})\ [\dot{*}]\,(r_2\,[-]\,k_2^{\mathcal{E}})\,[+]\,k_0^{\mathcal{E}} \qquad (\dot{*})$$

The $-$ and $+$ are the ordinary integer subtraction and addition operations respectively, and $[-]$ and $[+]$ are the corresponding operations in the ciphertext domain. The insight is that the semantics fits the theory, and it is as 'easy' for this hardware to implement the complex IEEE operation as any other, notwithstanding contemporary efforts to simplify and adapt floating point (for homomorphic encryption settings), c.f. [43–45]. It would also not be impractical to do in software (IEEE floating point emulation compiled encrypted performs well in tests).

The branch-if-equal instruction **beqf** $r_1\, r_2\, k_1^{\mathcal{E}}\, k_2^{\mathcal{E}}$ tests

$$(r_1\,[-]\,k_1^{\mathcal{E}})\ [\dot{=}]\,(r_2\,[-]\,k_2^{\mathcal{E}}) \qquad (\dot{=})$$

where $\doteq$ is the floating point comparison on floats via IEEE 754, and $[\doteq]$ is the corresponding test in the ciphertext domain, with $x^{\mathcal{E}}\,[\doteq]\,y^{\mathcal{E}}$ iff $x \doteq y$. The subtraction is as integers on the encoding, not floating point. The instruction is atomic, per (IIa), with embedded encrypted constants $k_i^{\mathcal{E}}$, $i = 1, 2$.

Representation of double precision 64-bit floats ('double') as 64-bit integers is also specified by IEEE 754. Let instructions that manipulate those (encrypted) be denoted by a **d** suffix. Registers are referenced in pairs in the instruction by naming the first of the pair only, its successor in the processor's register indexing scheme being understood as the second of the pair. The first of a pair contains an encrypted integer representing the 32 high bits of the IEEE 754 encoding of a double float, the second contains an encrypted integer encoding 32 low bits.

Definition B2. Let $\ddot{+}$, $\ddot{*}$, etc. denote double precision floating point addition, multiplication, etc. on IEEE 754 encodings of doubles as 64-bit (i.e., 2×32-bit) integers.

Let $[\ddot{*}]$ be the corresponding multiplication operation in the cipherspace domain on two pairs of encrypted 32-bit integers. Then the multiplication instruction on encrypted double precision floats has the following semantics, remembering registers are referenced in pairs for double length operations, and using the pairwise integer add and subtract operators of Definition 6:

$$r_0 \leftarrow (r_1\,[-^2]\,k_1^{\mathcal{E}})\ [\ddot{*}]\,(r_2\,[-^2]\,k_2^{\mathcal{E}})\,[+^2]\,k_0^{\mathcal{E}} \tag{$\ddot{*}$}$$

That satisfies (IIa-IId). It takes encrypted 64-bit double precision float operands in the (pair) registers r_1, r_2 and writes to the (pair) register r_0. The $k_i^{\mathcal{E}}$, $i = 0, 1, 2$ are encrypted 64-bit constants each embedded as two concatenated encrypted 32-bit constants in the instruction, which is written **muld** $r_0\, r_1\, r_2\, k_0^{\mathcal{E}}\, k_1^{\mathcal{E}}\, k_2^{\mathcal{E}}$.