

Safe Compilation for Encrypted Computing

Peter T. Breuer¹ and Simon J. Pickin²

¹ Hecusys LLC, Atlanta, GA, USA. Email: ptb@hecusys.com

² Universidad Complutense, Madrid, Spain. Email: spickin@ucm.es

Abstract. Encrypted computing is an emerging field in which inputs, outputs and intermediates are maintained in encrypted form in a processor, conferring security on user data against the operator and operating system as adversaries, which run unencrypted in the same machine. Systems that pass encrypted addresses to memory without decryption close a major attack vector and allow off-the-shelf memory to be used. But that makes memory unreliable from the program’s perspective, as the many different encryptions of a plaintext address access different memory locations that the program sees as the same with varying contents. A clever ‘obfuscating’ compiler solves the problem, opening up the field.

1 Introduction

This article describes compilation for *encrypted computing* that allows the processor to use addressing without decryption, opening up the emerging technology.

Encrypted computing means running on a processor that works profoundly encrypted in user mode, taking encrypted inputs to encrypted outputs via encrypted intermediate values in registers and memory. Other than that, the processor is conventional. In the privileged operator mode that the processor switches to in order to run exception handlers and other operating system code, the processor works unencrypted entirely as usual. The operator and operating system can access user data without hindrance, but it is always in the encrypted form they cannot read or write meaningfully, and security derives from that.

The arrangement aims to protect user data against attacks by the operator and operating system. The encryption key is embedded inaccessibly in the hardware, and the operator and operating system do not know it. The user knows the key and can provide encrypted inputs and interpret encrypted outputs – elsewhere –, but the operator cannot. That is firstly for lack of the key, and secondly by virtue of formal arguments in [1]. Several prototype processors for encrypted computing exist [2–6] (see Section 2). The fastest of these benchmarks like a 433MHz Pentium with a 1GHz base clock [7] and embeds AES encryption [8]. The basic principle is that the arithmetic embedded in the processor is modified, and that generates encrypted working [9]. To repeat:

The operator and operating system are the (potential) adversaries.

A *successful attack* is one that reads the plaintext of encrypted user data, or modifies it to order. The attack may have a statistical aspect, in which case it should succeed *more often than chance*.

A problem for processor designs is that addresses are data, hence exist in encrypted form, so conventional one-to-many encryption means there are many different encryptions of each plaintext address, up to 2^{96} with AES and 32-bit plaintext addresses plus random padding, for example. Those 2^{96} encrypted addresses must all be decoded by the processor to a unique location in memory so a program may read back what it has written. But equality of plaintext beneath the encryption can be physically checked by an adversary with access to the memory chips by seeing if when used for addressing the encrypted data accesses the same memory location, and that leads quickly to decryption. Locking the memory chips in place so ‘cold boot’ attacks [10, 11] (physically freezing the memory sticks to help preserve data while they are taken for analysis) cannot be used is no defense, as the operator can work the analysis programmatically: pattern all memory, do user mode writes using the encrypted data as addresses, then read memory to see where the writes broke the pattern.

Designs that defend against that rely on oblivious random access memory (ORAM) [12–15], which loses simplicity, and/or they assume some degree of ‘semi-honest’ behaviour by the operator, which compromises the aim, or they do not have computed addresses (the processor model is non-standard). Without ORAM, user data is still stored encrypted in memory because all user data enters and leaves the processor in encrypted form, so it is a pity to need ORAM just in order to physically protect the addresses that the processor must decrypt in order for memory to work correctly. It also does not on its own confer protection against the programmed form of the attack above. However, it turns out that there is an alternative: that the processor not decrypt addresses when passing them to memory, that a small translation unit in the processor remap the (say, 128-bit) encrypted addresses one-to-one on the fly down to a 32-bit range, and clever compilation take care of program correctness in that environment. Unencrypted addresses are never created, hence never exposed. The operator trying to compare encrypted data by using it as addressing only checks identity of the encrypted form of the address, not the plaintext address beneath the encryption.

The clue is given by ‘obfuscating’ compilation for encrypted computing. It is proved in [1] that with the appropriate set of primitive operations (see Section 2 for the set of *machine code instructions* used here), – all observations of and experiments with programs admit of 2^{32} different interpretations of the data beneath the encryption at any point in the program, given 32-bit plaintext data. Moreover, each interpretation is equally likely when the program has been generated by an ‘obfuscating’ compiler. That is described in [16] as:

An obfuscating compiler smooths out statistical biases in runtime traces.

It does that by stochastically varying the generated code from recompilation to recompilation so that the compiler’s variations at runtime statistically swamp contributions from human and other biases. To succeed in that, the compiler has to cope with as well as exercise control over arbitrary variations in data, and therefore in addressing, as addresses are data. It turns out that those compiler mechanisms are also able to cope with and control those 2^{96} encrypted variations of each 32-bit plaintext address that are naturally generated in the course of

running encrypted when the processor does no address decryption, with the result that programs run correctly. This paper explains the technique.

The only extra hardware required in the processor is the address translation lookaside buffer (TLB) unit referred to above, which remaps 128-bit encrypted addresses one-to-one to a 32-bit range as they are encountered during program running. The compiler also embeds instructions in the program code that free-up the mapping slots as opportunity arises. The capacity required for the unit is the maximum number of memory locations at any one time on which read-before-next-write is pending in the program (write all memory then read all memory is the worst case), which is never more than a program's natural memory footprint.

Technically, the compiler solution is for the general problem of *deterministic hardware aliasing* in computing platforms. Hardware aliasing [17] is where one address (here the plaintext address as perceived by the program) sporadically accesses different physical locations (here those picked out by the many different encrypted forms of the address), like a light switch that sometimes turns on the landing light and sometimes turns on the hall light. The 'deterministic' means it is not random but depends on some hidden aspect, here the padding and check bits that accompany the plaintext beneath the encryption. Processors are designed to produce repeatable results and those bits are calculated deterministically from the data and the sequence of operations it goes through. The features of this environment as far as the compiler is concerned are:

Axioms

1. A machine code copy instruction copies the physical bit sequence exactly, such that a copied encrypted address accesses the same location;
2. repeating the same sequence of operations produces an encrypted address that has exactly the same bit sequence and accesses the same location;
3. different plaintext addresses encrypt to different encrypted addresses.

Axiom 1 ('faithful copy') means the compiler can reliably save an encrypted address for later use after writing through it, and it will retrieve the written value. It must not be altered even by adding zero beneath the encryption, as that may alter the padding or check bits and hence the final encrypted address, which then fails to access the same memory location.

Axiom 2 ('repeatability') allows calculations so long as they are repeated exactly. That is vital because the machine code instruction to read or write a memory location takes a base address and introduces a displacement constant embedded in the instruction to get the final 'effective' address for the access. It is impossible to avoid that one extra operation, but it does not matter because it is repeated each time, reliably producing the same encrypted address each time.

Axiom 3 ('no confusion') guarantees that encryptions of different plaintext addresses do not step on each other. That is not so obvious considering the TLB frontend described above remaps the encrypted addresses and it is conceivable that it could release a target for reuse before the program has finished using the encrypted address that mapped to it. So the TLB does not release any mappings automatically. It is up to the compiler to insert instructions into the program that do the release (they reference the encrypted address, not the target).

Those abstractions allow computer architecture details to be elided and the layout of this paper is as follows. After some further background in Section 2, Section 3 explains how to compile following Axioms 1-3 to obtain code safe against hardware aliasing and therefore that works in the encrypted computing context. Section 4 incorporates the compiler obfuscation technique from [16] into the scheme. The compiler maximally varies the generated object code while always maintaining the same code structure as well as the same runtime trace structure, causing runtime data beneath the encryption to vary from nominal according to a determined obfuscation scheme. Section 5 shows how this works in practice. Short of formal proof of correctness, the only way to show a compiler works is to test by compiling and run programs that exercise each compiler construction, and the runtime trace from Ackermann function code is shown. That is the computable function of maximal complexity so it is difficult to ‘fix’. The trace of a Sieve of Eratosthenes for primes is also shown, with the array in memory, so there can be no trickery via running programs in registers only. Performance is not at issue and that programs work is the breakthrough.

Notation

Encryption of plaintext x is denoted by $\mathcal{E}[x]$ or $x^\mathcal{E}$, where \mathcal{E} is a one-to-many ‘nondeterministic function’, a function of x and extra hidden variables such as padding. Decryption of ciphertext ζ is denoted by $\mathcal{D}[\zeta]$, a function, with $\mathcal{D}[x^\mathcal{E}] = x$. The key k for encryption/decryption will be implicit when only one is involved, otherwise $\mathcal{E}[x, k]$ and $\mathcal{D}[\zeta, k]$. Equality (not identity) of ciphertexts $\chi = \zeta$ is defined as $\mathcal{D}[\chi] = \mathcal{D}[\zeta]$, so $x^\mathcal{E} = y^\mathcal{E}$ iff $x = y$, with $x^\mathcal{E} \neq y^\mathcal{E}$ iff $x \neq y$.

Operations on ciphertext will borrow the same names as on plaintext but in square brackets. Thus $x_1^\mathcal{E} [+] x_2^\mathcal{E} = \mathcal{E}[x_1 + x_2]$, meaning that $x_1^\mathcal{E} [+] x_2^\mathcal{E}$ may be calculated by decrypting the ciphertexts back to plaintexts x_1, x_2 , adding, then encrypting again. Whether the calculation is like that or not (the encryption may already possess that property), the abstraction is applicable.

Relations $x_1^\mathcal{E} [R] x_2^\mathcal{E}$ on ciphertexts denote the relation $x_1 R x_2$ on plaintexts.

2 Further Background

Several fast processors for encrypted computing are described in [5]. Those include the KPU [18], which runs encrypted on a 1 GHz clock with AES-128 [8] at the benchmark speed of a 433 MHz classic Pentium, and the slightly older HEROIC [4] which runs like a 25 KHz Pentium, embedding Paillier-2048 [19], as well as the recently announced CryptoBlaze [20], also using Paillier-2048 but $10 \times$ faster than HEROIC (it is not clear how many of those have working compilers).

The machine code instruction set defining the programming interface is important because a conventional instruction set is insecure against powerful insiders who may steal an (encrypted) user datum x and put it through the machine’s division instruction to get x/x encrypted, an encrypted 1. Then any desired encrypted y may be constructed by applying the machine’s addition instruction to

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

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

get $1 + \dots + 1$ encrypted. Via the order comparator instructions (testing $2^{31} \leq z$, $2^{30} \leq z$, ...) on an encrypted z and subtracting on branch, z may be obtained bitwise. That is a *chosen instruction attack* [1, 21]. An instruction set for encrypted computing must resist *algebraic* attacks like that, but the compiler must also be involved, else there would still be *known plaintext attacks* [22] based on the idea that human programmers intrinsically use values like 0, 1 more often than others. The compiler's job is to even out the statistics.

Necessary conditions, first described in [16], are shown in Box 1. Instructions must (i) execute atomically, or recent attacks such as Meltdown [23] and Spectre [24] against Intel are feasible, (ii) work with encrypted values or an adversary could read them, and (iii) be adjustable via onboard (encrypted) constants to offset by arbitrary deltas the runtime values beneath the encryption. The condition (iv) is for the security proofs in [1] and amounts to different padding or blinding factors for encrypted program constants and encrypted runtime values.

The compiler's job is to vary the encrypted constants (iii) embedded in the machine code instructions so all feasible trace variations are exercised *equiprobably*. [16] describes how an *obfuscation scheme* is generated. That is a set of vectors of *planned offsets from nominal for the data beneath the encryption per memory and register location, one vector for each machine code instruction*. A formal outline is that the compiler $C[-]^r$ translates an expression e that is to end up in register r at runtime to machine code mc and plans an offset δr in r :

$$C[e]^r = (mc, \delta r) \tag{1}$$

Let $s(r)$ be the value in register r in state s of the processor at runtime. The machine code mc changes state s to s' that holds a ciphertext in r whose plaintext value differs by δr from the nominal value $s(e)$ of e in s . That is:

$$s \xrightarrow{mc} s' \text{ where } s'(r) = \mathcal{E}[s(e) + \delta r] \tag{2}$$

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

Table 1. An instruction set for encrypted working.

<i>op.</i>	<i>fields</i>	<i>mnem.</i>	<i>semantics</i>
add	$r_0 r_1 r_2 \kappa$	add	$r_0 \leftarrow r_1 [+] r_2 [+] \kappa$
sub	$r_0 r_1 r_2 \kappa$	subtract	$r_0 \leftarrow r_1 [-] r_2 [+] \kappa$
mul	$r_0 r_1 r_2 \kappa_0 \kappa_1 \kappa_2$	multiply	$r_0 \leftarrow (r_1 [-] \kappa_1) [*] (r_2 [-] \kappa_2) [+] \kappa_0$
div	$r_0 r_1 r_2 \kappa_0 \kappa_1 \kappa_2$	divide	$r_0 \leftarrow (r_1 [-] \kappa_1) [\div] (r_2 [-] \kappa_2) [+] \kappa_0$
...			
mov	$r_0 r_1$	move	$r_0 \leftarrow r_1$
beq	$i r_1 r_2 \kappa$	branch	if b then $pc \leftarrow pc + i$, $b \Leftrightarrow r_1 [=] r_2 [+] \kappa$
bne	$i r_1 r_2 \kappa$	branch	if b then $pc \leftarrow pc + i$, $b \Leftrightarrow r_1 [\neq] r_2 [+] \kappa$
blt	$i r_1 r_2 \kappa$	branch	if b then $pc \leftarrow pc + i$, $b \Leftrightarrow r_1 [<] r_2 [+] \kappa$
bgt	$i r_1 r_2 \kappa$	branch	if b then $pc \leftarrow pc + i$, $b \Leftrightarrow r_1 [>] r_2 [+] \kappa$
ble	$i r_1 r_2 \kappa$	branch	if b then $pc \leftarrow pc + i$, $b \Leftrightarrow r_1 [\leq] r_2 [+] \kappa$
bge	$i r_1 r_2 \kappa$	branch	if b then $pc \leftarrow pc + i$, $b \Leftrightarrow r_1 [\geq] r_2 [+] \kappa$
...			
b	i	branch	$pc \leftarrow pc + i$
sw	$(\kappa) r_0 r_1$	store	$\text{mem}[r_0 [+] \kappa] \leftarrow r_1$
lw	$r_0 (\kappa) r_1$	load	$r_0 \leftarrow \text{mem}[r_1 [+] \kappa]$
jr	r	jump	$pc \leftarrow r$
jal	j	jump	$ra \leftarrow pc + 4$; $pc \leftarrow j$
j	j	jump	$pc \leftarrow j$
nop		no-op	

LEGEND

r - register index
 κ - encrypt. const.
 pc - prog. count reg.
 j - prog. count
 \leftarrow - assignment
 ra - return addr. reg.
 $\mathcal{E}[]$ - encryption
 i - prog. incr.
 r - register content
 $x^{\mathcal{E}}$ - encrypted val. $\mathcal{E}[x]$
 $x^{\mathcal{E}}[o] y^{\mathcal{E}} = \mathcal{E}[x \circ y]$
 $x^{\mathcal{E}}[R] y^{\mathcal{E}} \Leftrightarrow x R y$

An encrypted bit decodes
 beq/bne, ble/bgt, blt/bge.

The integer part of a generic instruction set architecture (ISA) satisfying (i-iv) above is shown in Table 1. It is adapted from the OpenRISC ISA v1.1 (<http://openrisc.io/or1k.html>), which has about 200 instructions. As well as the integer part, there are instructions for single and double precision integer operations, single and double floating point, and vector operations, all 32 bits long. Instructions access up to three 32 general purpose registers (GPRs), but one register operand may be replaced by a ('immediate') constant. Four 32-bit 'prefixes' precede a 32-bit instruction with 16 bits spare to provide $128=4 \times 28+16$ bits for one encrypted constant. That ISA covers the target instructions here.

3 Compiling around HW Aliasing in Encrypted Computing

The compiler attempts to implement the following basic strategy:

Each address that is written is saved for later read.

But it cannot do that. Though copying the address to stack^3 works (Axiom 1), the conundrum is that it is saved at an address that must also be saved.

Axiom 2 puts a backstop on the recursion, allowing an address that is calculated at compile time and recalculated at runtime (with the same result!) to be used. But a finite set of addresses cannot suffice for unbounded call depth. The real first problem to be solved is how to manipulate the *stack pointer* so addresses and other data can indeed be saved and recovered reliably from stack.

3.1 Stack Pointer 101

A standard function-entry code sequence decrements the stack pointer register **sp** by the amount needed for the stack frame³, then the standard function-exit

³ By 'stack' or '(stack) frame' is meant a runtime work area unique to a function call.

Table 2. Function call sequences

(a) standard/optimized	(b) with frame pointer	(c) complete
<pre>decrement sp ... increment sp return</pre>	<pre>copy sp to fp decrement sp ... copy fp to sp return</pre>	<pre>save old fp to 1 below sp copy sp to fp decrement sp ... copy fp to sp restore old fp from 1 below sp return</pre>

code sequence increments it again, as in Fig 2(a). That does not work in a hardware aliasing environment like encrypted computing, because the increment does not restore the exact bit sequence (the encrypted address) that was originally in the stack pointer register. Instead the caller gets back a different encryption of the same plaintext address and that accesses a different memory location.

But the *frame pointer* register **fp** can be used to save the stack pointer, and the latter restored from it as in Fig. 2(b). That is the typical unoptimized call sequence code produced by a compiler but default optimization ordinarily replaces it with the Fig. 2(a) code and frees up the frame pointer register for other uses. The GNU *gcc* compiler (for example) with **-fno-omit-frame-pointer** on the command line turns off optimization and produces the Fig. 2(b) code.

That is not yet perfect because the caller’s frame pointer register must still be saved from being trampled here, and later restored, as shown in Fig. 2(c). Saving below the caller’s stack pointer puts it in the callee’s stack frame, so the frame size must be over-stated for the decrement. As many as the compiler wants of the caller’s registers can be saved at the top of the callee’s frame.

Rather than decrement the stack pointer again for sub-blocks, the compiler reserves extra space for subframes within the function frame. Otherwise the same trick would have to be repeated for sub-blocks and accessing the function-level variables would need a chase through the sequence of stack pointers on the stack.

The final code in Fig. 2(c) works with hardware-aliasing. It sets up the function’s stack frame so it can be reliably addressed as $\sigma [+] d^{\mathcal{E}}$ from within the function, using the notation that will be standard in this document (see end of Section 1). Here d is a displacement between 0 and the frame size, provided as an encrypted constant $d^{\mathcal{E}}$ in a *load* or *store* machine code instruction, and σ is the encrypted address in the stack pointer register.

3.2 Accessing Variables

Given that setup for the stack pointer, accessing function local variables is simple. A word-sized local variable x is assigned a position d on the stack and the compiler issues a load instruction to read from there to register r :

```
lw r ( $d^{\mathcal{E}}$ )sp # load from offset  $d$  from  $sp$ 
```

The processor does addition $\sigma [+] d^\mathcal{E}$ in executing the instruction, but repeats that same calculation at every access, so by Axiom 2 of Section 1 the same sequence of bits for the encrypted address is produced every time, and it accesses the same spot in memory. To write the variable, a *store* instruction replaces *load*:

sw ($d^\mathcal{E}$)**sp** *r* # store to offset *d* from *sp*

For global variables, which reside at a compiler-decided address *a* in (heap) memory, the compiler offsets from the zero register **zer** instead of **sp**:

lw *r* ($a^\mathcal{E}$)**zer** # load from address *a*

The zero register contains a fixed zero value (not necessarily zero, indeed it is varied along with everything else for obfuscation as explained in Section 4). The encrypted effective address calculated by this instruction is always $\zeta [+] a^\mathcal{E}$, where ζ is the encrypted value in the **zer** register (notionally zero, but varied for each function) so the same memory location is always accessed.

The actual values of those encrypted addresses do not matter. They are just labels. They are all different as guaranteed by Axiom 3, and the processor remaps the 128-bit or more ciphertext of the encrypted addresses to 32-bit numbers that reference a physically backed area of memory, as discussed in Section 1.

3.3 Accessing Arrays

The real difficulties in an encrypted computing/hardware aliasing context arise with accessing arrays. One can access arrays in several ways, but whatever way one chooses one should stick to it and only use that, because in a hardware aliasing context like encrypted computing the answer for the encrypted address of the array element depends on the way it is calculated. One should not allow two ways of accessing the same array, because two different calculations for the encrypted address give two different answers.

The elements $\mathbf{a}[d]$ of array \mathbf{a} can in principle be accessed either via a load or store instruction with fixed displacement $d^\mathcal{E}$ from the encrypted array address α , or via a pointer that ranges through the array starting at α and steps through the elements until the one of interest is reached, at which point a load or store instruction with displacement zero from the pointer is used. The two calculations for the effective address are respectively $\alpha [+] d^\mathcal{E}$ and $\alpha [+] 1^\mathcal{E} [+] 1^\mathcal{E} \dots [+] 1^\mathcal{E} [+] 0^\mathcal{E}$. The calculations produce different bit sequences as encrypted addresses for the same plaintext address, so the two methods are incompatible and one ought to be used for each array. But in practice that is too restrictive. It is common, for example, both to step a pointer down through an array and step up through it.

We have accepted instead that general purpose array access is not going to be constant time in this kind of environment. For arrays of size N the compiler can provide access in $\log N$ time in a simple manner that works for all methods of access. In the encrypted computing context, it is even preferable that array access be nonconstant time, because in order to obfuscate which element is accessed, code should step through many, summing each in turn into an accumulator

multiplied by either 1 or 0 (encrypted). The observer not privy to the encryption cannot tell which multiplier is a 1 and which is a 0.

Linear complexity code will be presented. To read array element $\mathbf{a}[\mathbf{n}]$, index \mathbf{n} is tested against $0, \dots, N-1$ in turn. The code produced is equivalent to:

```
(n == 0)?a[0]:
(n == 1)?a[1]:
...
```

The equality tests are insensitive to the particular encryption $n^\mathcal{E}$ of plaintext number n in the variable \mathbf{n} . The processor evaluates $n^\mathcal{E} [==] 0^\mathcal{E}$ to $n==0$, and branches accordingly. The encrypted address passed to memory for $\mathbf{a}[n]$ is always $\alpha [+] n^\mathcal{E}$. Here n is the displacement from the base of the array and α is the encrypted address for the lowest array element $\mathbf{a}[0]$. That is $\alpha = \sigma [+] k^\mathcal{E}$, where k is the allocated position of the array on the stack and σ is the encrypted address in the stack pointer register. So the address passed to memory for $\mathbf{a}[n]$ is $\sigma [+] k^\mathcal{E} [+] n^\mathcal{E}$ and the machine code sequence generated by the compiler to do the read is:

```
addi r sp kε # add kε to σ from sp to form α in r
lw r (nε)r # load from address α[+]nε with α in r to r (3)
```

where the **addi** instruction adds its constant operand to the register as indicated.

Improving this code to log N complexity entails using a binary tree structure. Code for writing to $\mathbf{a}[\mathbf{n}]$ follows the same pattern, with store instead of load.

The same code structure works for access via a pointer \mathbf{p} , but the compiler needs to know which array it points into. The problem is that pointers in C can point anywhere at runtime and the compiler can hardly ever predict where, so the equality tests above must run over all 2^{32} possible plaintext addresses for each access. That is impractical so we have tightened the type system of C so the pointer is declared along with the name of a (possibly overlarge) array \mathbf{a} into which it will definitely point at runtime:

```
restrict a int *p
```

The **restrict** keyword selects the target array for the pointer. This means a certain amount of porting has to be done for existing code, marking out global areas into which pointers can point. It generally means declaring a global array from which objects of the kind pointed to are allocated from, or declaring a function as interior to another function where the target of the pointer is defined as a local. As the new pointer type is narrower than the original and (ideally) we make no semantic changes, confidence in type safety should be increased.

Then the code generated does lookup via pointer \mathbf{p} like this:

```
(p == a+0)?a[0]:
(p == a+1)?a[1]:
...
```

It is insensitive to the way the pointer \mathbf{p} is calculated in a dereference $*\mathbf{p}$ just as the code for $\mathbf{a}[\mathbf{n}]$ is insensitive to the way \mathbf{n} is calculated. The code can

similarly be made over to $\log N$ complexity with a binary tree lookup structure, and converted to write by replacing load instructions at the leaves with stores. These constructions make pointer access the same as access via array index.

3.4 Long Types

Records with named fields ('struct' in C) are treated by the compiler as arrays and the field name is translated to an array displacement. The declaration

```
struct { int a; int b; } x
```

declares \mathbf{x} with two named fields, \mathbf{a} and \mathbf{b} , each one word wide. It occupies two words on the stack at displacements k and $k + 1$ respectively from the stack pointer. The compiler generates accesses to $\mathbf{x.a}$ and $\mathbf{x.b}$ just as it would for any local variables situated there as described in Section 3.2. Source code attempting to access the fields of the struct as though it were an array still works, which helps in porting C code that uses this type-unsafe but commonly used pattern.

Long atomic types such as **double** (a 64-bit float encoded as two encrypted 32-bit integers following the IEEE 754/ISO 60559 standard) are also treated as arrays with the compiler generating single word load and store instructions to access separately the (encrypted) high and low words of the long type.

But platforms for encrypted computing also can have double-word load and store instructions that fetch/write two encrypted words at once:

```
ld r (kε)sp # double word load
```

The effective address of the first word is $\sigma [+] \mathcal{E}[k]$, σ being the encrypted address in the stack pointer register. Registers are indexed in pairs for this instruction, and the second of the pair is loaded up by it with a second encrypted word. But that comes from the address $(\sigma [+] \mathcal{E}[k]) + 1$ in memory, not $\sigma [+] \mathcal{E}[k + 1]$. It has to be so because the memory is by design not privy to the encryption and so cannot decrypt the effective address of the first word and add one and encrypt again to get the address of the second word.

Even if it could do so, the best it could produce is $\sigma [+] \mathcal{E}[k] [+] \mathcal{E}[1]$ since $\sigma [+] \mathcal{E}[k]$ is what is given (that is possible if the encryption is partially homomorphic so the encrypted addition can be done without the key) and that is not the same encrypted value as $\sigma [+] \mathcal{E}[k + 1]$, which is the effective address of the second word of the double accessed like an array. A second processor with the key would be needed within the memory management unit to which to offload the calculation checking $\sigma [+] \mathcal{E}[k] [+] \mathcal{E}[1]$ is an encryption alias of $\sigma [+] \mathcal{E}[k + 1]$.

The conclusion is that either double word instructions must be used all the time, including for single words (there would follow an arithmetic operation to extract the encrypted 32-bit half required), or the single word instructions must be used all the time. We in our prototype compiler have chosen to never invoke double word, just single word instructions.

3.5 Short Types

The difficulty in accessing the second word of a **double** in memory translates to a difficulty with the individual bytes of a word. For indexed access to the characters of a string **a**, a compiler should generate a sequence of arithmetic operations that splits the character index **i** into a word index **d** and an offset **j** for the character within the word, equivalent to this code:

```
d = i/4;
j = i%4;
(a[d / 256j) % 256 # jth char of dth word
```

In our own prototype compiler we have preferred to avoid this complication and pack characters one to a word, at the cost of an inefficient use of memory for strings (the upper bits are random). Then no arithmetic as above is required.

4 Integrating Compiling and Obfuscation

Although hardware aliasing takes place ubiquitously from the point of a program running in an encrypted computing environment, that is not the only hurdle for a compiler. As described in Section 1, the compiler has to vary code and data to swamp out human programming influences that could lead to statistically based attacks. If zero is the most common data item, it pays to mount a *statistical known plaintext attack* [22] supposing any given encrypted datum is zero. The technique (for integers only) is set out in [16] and it is to vary offsets from the nominal values in the plaintext beneath the encryption in the content of memory and registers, as described in (2) of Section 1. That fits with the address control.

4.1 Varying Address Displacement Deltas

Instead of generating a load instruction to read from a variable at position n on the stack as in (3) of Section 3.2:

```
lw r (nε)sp # load from offset n from sp
```

the compiler will vary the *displacement constant* n to Δ :

```
lw r (Δε)s # load from offset n from sp
```

Here Δ is a previously chosen random number and the register s has been pre-set with the value $\sigma [+] n^\epsilon [-] \Delta^\epsilon$ to accommodate this, where σ is the encrypted address in the stack pointer register. The effective address passed to memory is:

$$\sigma [+] n^\epsilon [-] \Delta^\epsilon [+] \Delta^\epsilon$$

and is always the same when that calculation/instruction sequence is repeated and always accesses the same memory location (Axiom 2, Section 1). The compiler has to ensure separately that Δ^ϵ is always the same for the same n . It maintains a vector $\mathbf{\Delta}^\epsilon$ indexed by stack location n , as well as a similar vector $\mathbf{\Delta}_Z^\epsilon$ for the heap, and the load instruction above has $\Delta^\epsilon = \mathbf{\Delta}^\epsilon n$.

4.2 Varying Content Deltas

The stack pointer \mathbf{sp} does not contain the value it notionally should have but is offset by a random delta and that is true of every register at every point in the code. The compiler maintains a vector $\delta_R^\mathcal{E}$ of offsets $\delta_R^\mathcal{E} r$ in each register r , varying it as it makes its pass through the code.

It also maintains a vector $\delta_Z^\mathcal{E}$ of delta offsets of stack contents indexed by stack location n , and a vector $\delta_Z^\mathcal{E}$ for the delta offsets of heap contents indexed by heap location n . Code for accessing the n th location on the stack is:

```

addi r sp  $k^\mathcal{E}$  # where  $k = n - \delta_R \mathbf{sp} - \Delta n$ 
lw r  $(\Delta^\mathcal{E} n)_r$  # read  $n$ th location on stack

```

and the effective address received by memory is

$$\sigma [+] \mathcal{E} [n - \delta_R \mathbf{sp} - \Delta n] [+] \Delta^\mathcal{E} n$$

where the stack pointer register \mathbf{sp} contains the encrypted address $\sigma = sp^\mathcal{E} [+] \delta_R^\mathcal{E} \mathbf{sp}$ differing in $\delta_R \mathbf{sp}$ from the nominal plaintext value sp . Summing, the address has the plaintext value $sp + n$. The calculation for the effective address is the same every time so the bit sequence passed to memory is the same every time, by Axiom 2 of Section 1. A write just replaces the load instruction by store.

The δ and Δ values are changed randomly by the compiler just before every point in the code where a write occurs. A result in work presently under review [25] measures the variability in a runtime trace in terms of its entropy viewed as a stochastic random variable over recompilations of the same source code:⁴

Theorem 1. *The entropy in a trace over recompilations is the sum of the entropies of every instruction that writes that appears in it, counted once each.*

An instruction has entropy inasmuch as its effect can be varied by the compiler from recompilation to recompilation by changing the constants embedded in it. Machine code instructions like load and store that copy from one place to another do not contribute entropy to the trace except as they may write to different places and the address displacement deltas (Section 4.1) contribute entropy there. But though the compiler varies the contribution Δn in the **addi** instruction in the load sequence above, potentially supplying 32 bits of entropy (say), Δn is fixed through reads and only varied at writes, so it must be ‘old news’ when the sequence runs and contributes no entropy.

The compiler’s job is to do everything it can to maximize the trace entropy:

Theorem 2. *Trace entropy is maximized when the compiler varies every instruction that writes to the maximal extent possible (i.e., with flat distribution). ($\overline{\mathcal{H}}$)*

(again from [25]). That provides a stochastic setting in which an attacker cannot be sure what the numerical value of the runtime plaintext should be, even in

⁴ The entropy is the expectation $E[-\log_2 p(X)]$ for variable X with distribution $p(X)$, and it expresses roughly a number of independent (binary) degrees of freedom.

Table 3. Code patterns

(a) while loop	(b) conditional
<pre> start: mc_e # compute e in r beqz r end # goto to end if r zero mc_s # compute s b start # goto start end: </pre>	<pre> start: mc_e # compute e in r beqz r else # goto to else if r zero mc_{s_1} # compute s_1 b end # goto end else: mc_{s_2} # compute s_2 end: </pre>

terms of a statistical tendency. The principle is that the compiler provides an extra, additive, uniformly distributed input at each instruction where it is able to contribute ($\overline{\mathcal{H}}$). Via Shannon’s Law [26], the resulting plaintext data distribution beneath the encryption at runtime has maximal entropy over its 32 bits at any point. That means data beneath the encryption in the trace is no more vulnerable to statistical plaintext attack than random (encrypted) data is, given that the encryption is independently secure (this paraphrases the proof in [7]).

All that is required per Theorem 2 is to confirm that each of the compiler’s constructions vary maximally every instruction that writes that it outputs ($\overline{\mathcal{H}}$).

4.3 Doing It for Loops and Conditionals

Just two compiler constructions will be detailed here. Loops (**while**, **for**, backward **goto**, etc.) reduce the available entropy in traces and this is why. Let the statement compiler $C[-]$ produce code mc from statement s of the source language, altering the combined database $D = (\Delta, \Delta_Z, \delta, \delta_Z, \delta_R)$ to D^s . Let D be of type DB, statements s of type Stat, machine code mc of type MC, then the type statement for the compiler is (pairs (D, x) are written $D : x$ here for legibility):

$$\begin{aligned}
 C[- : -] &:: \text{DB} \times \text{Stat} \rightarrow \text{DB} \times \text{MC} \\
 C[D : s] &= D^s : mc
 \end{aligned} \tag{4}$$

(compare (1) for expressions e). Compiling **while** e s means producing code mc constructed from code mc_e for e , mc_s for s , with the pattern in Table 3(a). The compiler produces first D^e then D^s , dropping out codes mc_e , mc_s , in that order:

$$D \xrightarrow[\underset{mc_e}{\sim}]{} C[-:e]^r D^e \xrightarrow[\underset{mc_s}{\sim}]{} C[-:s] D^s \tag{5}$$

But the pattern in Table 3(a) does not quite work, because it does not reestablish the deltas at loop start and second time through the loop goes wrong. Extra *trailer* instructions are needed at the end of the loop body after mc_s . This trailer reverts r to its initial delta $\delta_R^E r$ from the final delta $\delta_R^s r$ off nominal:

$$\text{addi } r \ r \ k^E \quad \# \text{ add } k = \delta_R r - \delta_R^s r \tag{6}$$

Trailer instructions also reestablish the start-of-loop address displacements. The trick is to read with the end-of-loop displacement $\Delta^s n$ then write with the start-of-loop displacement Δn , sandwiching the revert above:

```

addi t0 sp jε      # j = n - δR sp - Δs n
lw t0 (Δsε n)t0    # load nth stack location
addi t0 t0 kε      # modify by k = δn - δs n
addi t1 sp lε      # l = n - δR sp - Δ n
sw (Δε n)t1 t0    # store nth stack location

```

(7)

It does not matter if $k^\varepsilon = \mathcal{E}[\delta_R r - \delta_R^s r]$ or, e.g., $\delta_R^\varepsilon r [-] \delta_R^s r$ is used because even used as an address it is immune to differences in calculation given the compiler constructions of Section 3. The point is that the numbers j, k, l are completely determined by the compiler's earlier choices (of deltas) for instructions in the loop. It is impossible to get to the trailer instructions without traversing the loop and executing those earlier instructions, so these numbers are always 'old news' and they introduce no more entropy into the trace. Apart from those, which cannot be made to import more entropy, each instruction that writes in mc is in one of mc_e or mc_s , and those (by induction) may be varied maximally by the compiler, confirming $(\overline{\mathcal{H}})$ as required for Theorem 2.

Resynchronization is needed wherever two control paths join and after an if-then-else in particular: the final deltas in the two branches must be equalized and the compiler emits trailer instructions for that. Code mc for **if** s_1 **else** s_2 has the pattern in Table 3(b) and the compiler emits mc_e, mc_{s_1}, mc_{s_2} in order:

$$D \overset{C[-:e]^r}{\underbrace{\hspace{1.5cm}}} D^e \overset{C[-:s_1]}{\underbrace{\hspace{1.5cm}}} D^{s_1} \overset{C[-:s_2]}{\underbrace{\hspace{1.5cm}}} D^{s_2} \quad (8)$$

But that code also does not work as-is. Trailer instructions (6-7) must be placed after mc_{s_2} to set the final deltas in the 'else' branch equal to those in the 'then' branch. If the 'else' branch is executed first then the trailers import to the trace the entropy from the final choices of delta in the (not yet executed) 'then' branch but they cannot be varied further to introduce more entropy than that. If the 'then' branch is executed first then the trailers import no more entropy into the trace when they do execute because their numbers are determined by instructions executed earlier, but they still could not have been varied to introduce more entropy than that as they are determined by the requirements. So the conclusion is that the trailers do introduce all the entropy they can, and by induction the instructions in mc_e, mc_{s_1}, mc_{s_2} do all they can too, confirming $(\overline{\mathcal{H}})$. Similarly for other compiler constructions (not shown). Then by Theorem 2:

Corollary 1. *Trace entropy is maximized by the compiler constructs.*

The principle $(\overline{\mathcal{H}})$ guides design. It dictates that every entry in an array must have a separate delta in the obfuscation database D , rather than all the entries sharing the same delta. A write to an entry must be followed by a change of delta, as it is an opportunity for the compiler to vary the writing instruction. If the delta were shared, then all entries in the array would have to be updated to the new delta, but then the instructions in the 'write storm' to the other entries would not introduce more entropy, as their delta is already known.

Table 4. Trace for Ackermann(3,1)

PC	instruction		trace updates
...			
35	addi t0 a0	-86921031 ^E	t0 ← -86921028 ^E
36	sub t1 zer zer	-327157853 ^E	t1 ← -327157853 ^E
37	beq t0 t1 2	240236822 ^E	
38	sub t0 zer zer	-1242455113 ^E	t0 ← -1242455113 ^E
39	b 1		
41	sub t1 zer zer	-1902505258 ^E	t1 ← -1902505258 ^E
42	xor t0 t0 t1	-1734761313 ^E	1242455113 ^E 1902505258 ^E t0 ← -17347613130 ^E
43	beqz t0 9	-1734761313 ^E	
53	addi sp sp	800875856 ^E	sp ← 1687471183 ^E
54	addi t0 a1	-915514235 ^E	t0 ← -915514234 ^E
55	sub t1 zer zer	-1175411995 ^E	t1 ← -1175411995 ^E
56	beq t0 t1 2	259897760 ^E	
57	sub t0 zer zer	11161509 ^E	t0 ← 11161509 ^E
...			
143	addi v0 t0	42611675 ^E	v0 ← 13 ^E # result
...			
147	jr ra		
	STOP		

Legend	
<i>op. fields</i>	<i>semantics</i>
addi r0 r1 κ	r0 ← r1[+]κ
...	(see Table 1)
<i>register</i>	<i>semantics</i>
a0,a1,...	function argument
pc	program counter
ra	return address
sp	stack pointer
t0,t1,...	temporaries
v0,v1,...	return value
zer	null

5 Examples

Our own prototype C compiler <http://sf.net/p/obfusc> covers ANSI C and GNU C extensions, including statements-as-expressions and expressions-as-statements, gotos, arrays, pointers, structs, unions, floating point, double integer and floating point data. It is missing **longjmp**, computed goto, efficient strings (**char** is the same as **int**), and global data shared across different files (a linker issue).

A trace⁵ of the Ackermann function⁶ [27] compiled by that compiler is shown in Table 4. The trace illustrates how the compiler's variation of the delta offsets for register content through the code results in randomly generated constants embedded in the instructions and randomly offset runtime data. The Ackermann function is the toughest test possible of the correctness of the compiler's strategies for function call, assignment and conditionals. It is not realistically feasible to get the correct result without being perfect all the way through.

Running a Sieve of Eratosthenes program⁷ for primes shows that arrays work too. How memory access is affected by address displacement constants is visible in the trace in Table 5, which also shows the padding on the data as well as the (decrypted, plaintext) data itself. One stack read has been marked in red. The address base and displacement from base in the are in blue so it can be seen how the base register is first decremented arbitrarily low by the compiler and then compensated high in the displacement in the load. The variation is arbitrary but

⁵ Initial and final content offset deltas are set to zero here, for readability.

⁶ 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)); }.

⁷ 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; } .

Table 5. Trace for sieve showing hidden bits in data (right). Stack read instructions in red, address base and address displacement in blue.

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

exactly the same sequence of instructions must be repeated at every access, as the padding (shown) would differ otherwise, and it forms part of the effective address that is passed to memory.

6 Conclusion

This paper shows how to compile for encrypted computing environments that pass encrypted addresses undecoded to memory, resulting ‘hardware aliasing’ from the perspective of a program, due to the one-to-many nature of good encryption and the many different encryptions of a single plaintext address.

That is an advance that allows ordinary RAM to be used as the memory device in processors for encrypted computing, validating the emerging technology. That aims to provide security for user programs, which run encrypted, against the operating system, which runs unencrypted but otherwise with unlimited access and privileges.

The technique adapts obfuscating compilation, which varies and controls offsets from nominal in the content of each memory location at runtime, to control also offsets in the addresses.

The compiler constructions described in this paper generate log or linear complexity array accesses. Constant complexity would be possible but at the price of making index- and pointer-based access incompatible, which would make porting existing source codes to the platform much more difficult.

References

1. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: On security in encrypted computing. In Naccache, D., et al., eds.: Proc. 20th Int. Conf. Info. Comms. Sec. (ICICS’19). Number 11149 in LNCS, Cham, Springer (October 2018) 192–211

2. Fletcher, C.W., van Dijk, M., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: Proc. 7th ACM Scalable Trusted Comp. Workshop (STC'12), New York, ACM (2012) 3–8
3. Tsoutsos, N., Maniatakos, M.: Investigating the application of one instruction set computing for encrypted data computation. In Gierlichs, B., Guilley, S., Mukhopadhyay, D., eds.: Proc. Sec., Priv. Appl. Cryptog. Eng. (SPACE'13). Springer, Berlin/Heidelberg (2013) 21–37
4. Tsoutsos, N., Maniatakos, M.: The HEROIC framework: Encrypted computation without shared keys. *IEEE Trans. CAD Integ. Circ. Sys.* **34**(6) (2015) 875–888
5. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: Superscalar encrypted RISC: The measure of a secret computer. In: Proc. 17th Int. Conf. Trust, Sec. Priv. Comp. Comms. (TrustCom'18), CA, USA, IEEE Comp. Soc. (August 2018) 1336–1341
6. Breuer, P.T., Bowen, J.P.: (un)encrypted computing and indistinguishability obfuscation. CoRR [abs/1811.12365](https://arxiv.org/abs/1811.12365) (2018) (Extended abstract) Princip. Sec. Compil. Track (PRiSC'19), 46th ACM SIGPLAN Symp. Princip. Prog. Lang. (POPL'19).
7. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: The secret processor will go to the ball: Benchmark insider-proof encrypted computing. In: Proc. 3rd Work. Safety & Security aSSurance Critical Infrastructures Protection (S4CIP'18) / 3rd Euro. Symp. Security and Privacy Workshops (EuroS&PW'18), CA, USA, IEEE comp. soc. (April 2018) 145–152
8. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Springer, Berlin/Heidelberg (2002)
9. Breuer, P.T., Bowen, J.P.: A fully homomorphic crypto-processor design: Correctness of a secret computer. In Jürjens, J., Livshits, B., Scandariato, R., eds.: Proc. 5th Int. Symp. Eng. Sec. Soft. Sys. (ESSoS'13). Number 7781 in LNCS, Berlin/Heidelberg, Springer (February 2013) 123–138
10. Simmons, P.: Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In: Proc. 27th Ann. Comp. Sec. Appl. Conf. (ACSAC'11), New York, NY, ACM (2011) 73–82
11. Gruhn, M., Müller, T.: On the practicability of cold boot attacks. In: 8th Int. Conf. Availability, Reliability, Sec. (ARES'13). (September 2013) 390–397
12. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: Proc. 22nd Ann. ACM Symp. Th. Comp. (1990) 514–523
13. Ostrovsky, R., Goldreich, O.: Comprehensive software protection system (June 16 1992) US Pat. 5,123,045.
14. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)* **43**(3) (1996) 431–473
15. Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In Sahai, A., ed.: Proc. 10th Th. Cryptog. Conf. (TCC'13). Volume 7785 of LNCS. Springer, Berlin/Heidelberg (March 2013) 377–396
16. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: On obfuscating compilation for encrypted computing. In Samarati, P., Obaidat, M.S., Cabello, E., eds.: Proc. 14th Int. Conf. Security and Cryptography (SECRYPT'17). Volume 4., Portugal, INSTICC, SCITEPRESS (July 2017) 247–254
17. Barr, M.: Ch. 6, Memory. In Oram, A., ed.: *Programming Embedded Systems in C and C++*. 1st edn. O'Reilly & Associates, Inc., Sebastopol, CA (1998) 64–92
18. Breuer, P.T., Bowen, J.P., Palomar, E., Liu, Z.: A practical encrypted micro-processor. In Callegari, C., et al., eds.: Proc. 13th Int. Conf. Sec. Cryptog. (SECRYPT'16). Volume 4., Portugal, SCITEPRESS (July 2016) 239–250
19. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Proc. EUROCRYPT'99. Adv. Cryptol., Springer (1999) 223–238

20. Irena, F., Murphy, D., Parameswaran, S.: Cryptoblaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support. In: Proc. 23rd Asia S. Pac. Design Autom. Conf. (ASP-DAC), Los Alamitos, CA, USA, IEEE (2018) 702–708
21. Rass, S., Schartner, P.: On the security of a universal cryptocomputer: The chosen instruction attack. *IEEE Access* **4** (2016) 7874–7882
22. Biryukov, A.: Known plaintext attack. In van Tilborg, H.C.A., Jajodia, S., eds.: *Encyclopedia of Cryptography and Security*. Springer, Boston, MA (2011) 704–705
23. Lipp, M., et al.: Meltdown. *ArXiv e-prints* (January 2018)
24. Kocher, P., et al.: Spectre attacks: Exploiting speculative execution. *ArXiv e-prints* (2018)
25. Breuer, P.T.: An information obfuscation calculus for encrypted computing. *Cryptography ePrint Archive*, Report 2019/084 (January 2019)
26. Shannon, C.E.: A mathematical theory of communication. *Bell System Technical Journal* **27**(3) (October 1948) 379–423
27. Sundblad, Y.: The Ackermann function. a theoretical, computational, and formula manipulative study. *BIT Num. Math.* **11**(1) (March 1971) 107–119