International Conference on Computational Science, ICCS 2013

An Open Question on the Uniqueness of (Encrypted) Arithmetic

Peter T. Breuer[a,*], Jonathan P. Bowen[b]

[a]*University of Birmingham, Edgbaston, Birmingham, UK*
[b]*London South Bank University, London, UK*

Abstract

We ask whether two or more images of arithmetic may inhabit the same space via different encodings. The answers have significance for a class of processor design that does all its computation in an encrypted form, without ever performing any decryption or encryption itself. Against the possibility of algebraic attacks against the arithmetic in a 'crypto-processor' (KPU) we propose a defence called 'ABC encryption' and show how this kind of encryption makes it impossible for observations of the arithmetic to be used by an attacker to discover the actual values. We also show how to construct such encrypted arithmetics.
Keywords: arithmetic, cipherspace, crypto-processor, encryption, processor design

1. Introduction

A KPU [1] is a processor that works natively on encrypted data. The generic term for such is a 'general purpose crypto-processor' [2, 3, 4], aka a 'secret computer', and a KPU is the particular RISC [5] design that appears in [1]. No encryption or decryption ever takes place in a KPU, but data is nevertheless always maintained in encrypted form within registers and memory as the program is executed. Not being in possession of the encryption key, a physically present observer cannot in principle understand the computation in progress, which makes a KPU-based design a candidate for future cloud computation infrastructures; ideally each thread of computation uses its own encryption, thus isolating threads with different owners from each other and the platform operator by virtue of their uniquely different encryptions. This paper asks a computational question about how many different arithmetics there are of a form that makes them suitable for use as the encrypting 'kernel' inside a KPU. On the answer hinge questions of design, security and the number of simultaneous differently encrypted threads of computation that may run.

In explanation, the 'trick' behind the KPU design is that its arithmetic is altered with respect to the standard. A processor performs arithmetic by means of piece of hardware called an 'ALU' ('arithmetic logic unit') inside it and this is the one physical element that is altered in a KPU with respect to a standard CPU. The paper [1] shows that the physical states that arise in memory and registers when a program runs in a KPU are encryptions of the states that would arise when the same program is run in an ordinary CPU, given certain provisos. The provisos are that the running machine code program is 'type-safe' with respect to keeping encrypted data separate from unencrypted data. Another earlier paper [6] sets out a typed assembly language for a KPU such that any correctly compiled (i.e. well-typed) program is type-safe in this sense, and thus runs 'correctly' in that the states obtained by the processor are precisely the states expected in a standard CPU, modulo encryption.

---

* Corresponding author. Tel.: +44-785-225-9426 .
E-mail address: ptb@cs.bham.ac.uk,ptb@ieee.org.

The standard ALU, when the bit-pattern for the number 1 is presented on its first input, the number 1 on its second input, and the pattern signifying 'add' is presented on its control input, produces the bit-pattern for the number 2 on its output, thus implementing '1+1=2'. An ALU implements the arithmetic operations of addition, subtraction, multiplication, division, as well as the bit-wise logical operations AND, OR, etc. In a KPU the addition operation in the altered ALU may affirm that '43+43=5740346', and that that is an encrypted version of '1+1=2' is known only to the owner of the computation.

Denoting encryption by $\mathcal{E}$, and the addition operation inside the KPU's altered ALU by $\boxplus$, the altered operation is designed to satisfy the *homomorphism condition*:

$$\mathcal{E}(x) \boxplus \mathcal{E}(y) = \mathcal{E}(x + y)$$

with respect to the standard addition, for arbitrary integer inputs x, y. Similarly for the other arithmetic operations in the altered ALU. This kind of encryption is called 'fully homomorphic' [7], and another way of looking at KPU design is that its hardware is constructed so as to make a given block encryption $\mathcal{E}$ (with blocksize the physical size of an encrypted data word) into a fully homomorphic encryption. A practically useful fully homomorphic encryption is such that one can give away the function '$\boxplus$' to all users, and not give away the encryption key. So, for example, if a user posseses some digital money, say $10, protected by the encryption, and another user posseses $1 protected by the same encryption, then the first can give the $1 to the second and the second can add the $1 to the $10 under the encryption using the published '$\boxplus$' function. No decryption is required.

In truth, every block encryption ever thought of is 'fully homomorphic' in the abstract - it is just that there is no construction known of its '$\boxplus$' operation other than 'decrypt, add, encrypt'. Clearly one cannot publish that construction, since it means giving away the general decryption function. But in a KPU the altered ALU embodies the '$\boxplus$' function and in principle it can be given away to everyone. That is, everyone is free to inspect it and even play with it and we believe that the secret of decryption is not compromised thereby. We will substantiate that claim below.

This paper is structured as follows. The next section discusses how 'algebraic' attacks may be constructed against the arithmetic in a KPU, and proposes a defence, called 'ABC encryption'. The section shows how ABC encryption makes it impossible for an attacker to apply a formula to some observations of arithmetic within the KPU in order to deduce what is the encrypted value that corresponds to some particular unencrypted value that is known a priori. The following section shows how encrypted arithmetics satisfying the ABC-scheme may be constructed. It turns out that the same ABC-encrypted arithmetic may embed two (or more) interpretations of ordinary arithmetic at once. This provides a prima facae defence against an attacker with part or all of the encrypted arithmetic already in hand; the attacker cannot know which of the interpretations is in use, since the difference lies in the users own belief. The section poses the 'how many' question in order that it may be explored by those with the appropriate computing resources; it also provides a survey of the few known answers.

2. Attacks, defences, and ABC encryption

Note that arithmetic is carried out modulo 2^{32} in a 32-bit ALU, so that 1024*1024*1024*4 = 0, for example, giving rise to corresponding equalities such as

$$\mathcal{E}(1024) \boxtimes \mathcal{E}(1024) \boxtimes \mathcal{E}(1024) \boxtimes \mathcal{E}(4) = \mathcal{E}(0)$$

in the altered ALU in a KPU. Such 'algebraic' tautologies give rise to avenues of attack on the encryption. In principle, an attacker can deduce what the encryption of 0 is, for example, by creating the encryption of an even number by adding some encryption $\mathcal{E}(x)$ (presumably obtained by licit observation; the unencrypted value x is not known to the attacker) to itself using the altered ALU, giving $\mathcal{E}(2x)$, then doubling that again and again using the altered ALU until the result shows no further change, which will be $\mathcal{E}(0)$.

Other equalities allow one to search for and recognize $\mathcal{E}(1)$, and then, by adding using the altered ALU again and again, to create $\mathcal{E}(2) = \mathcal{E}(1) \boxplus \mathcal{E}(1)$, $\mathcal{E}(3) = \mathcal{E}(2) \boxplus \mathcal{E}(1)$, etc, until the attacker has a complete encryption codebook for the 32-bit integers.

Non-determinism in the encryption

Unfortunately for an attacker, a single codebook is never quite enough, because the encryption 'function' is never deterministic in practice, which in this case means that the 'cipherspace' requires more than 32 bits; 40, 48, 56 or 64 bits are typical choices. That is, there are usually *many* different encodings of '1' in the same encryption (perhaps 43, 101, 97687, ...), all of which mean '1'. Thus, an attacker cannot expect to recognize conditions such as 'the result [of repeated doubling] shows no further change', as proposed above.

The non-determinism need not be stochastic; that is, repeating the same calculation through the ALU should give the same answer every time. But doing the same calculation a different way, say by adding '1+1+1' instead of '1+2' may give rise to a different encryptions of the expected answer '3', so that the two encryptions will not be recognized by an attacker as representing the same unencrypted value.

Algebraic attacks

However, non-determinism is no defence when the ALU contains an implementation of division. Then it suffices to divide any encrypted value by itself using the altered ALU to get an encryption of 1:

$$\mathcal{E}(x) \boxminus \mathcal{E}(x) = \mathcal{E}(x/x) = \mathcal{E}(1)$$

for $x \neq 0$. Once one has an encryption of 1, then one has an encryption of 2 as $\mathcal{E}(1) \boxplus \mathcal{E}(1) = \mathcal{E}(2)$, and so on, until a codebook is constructed. Once one encryption of every 32-bit number has been discovered in this way, then the attacker may try different calculations that should give the same unencrypted result in order to discover yet more encryptions of the same numbers.

Even if division is not implemented in the hardware, then one may expect that it is implemented in software. The attacker may locate a division subroutine in memory and use it to do encrypted division with. If the division subroutine cannot be found, then life is more awkward for the attacker, but x/x is not the only formula that returns a known constant. It is the case that $x - x$ always returns 0, for example, so an attacker may find an encryption of 0 by computing $\mathcal{E}(x) \boxminus \mathcal{E}(x) = \mathcal{E}(0)$ with the altered ALU. If the attacker has finally located all the possible encryptions of 0, then a list of all the encryptions of numbers x which are not 'divisors of zero' may be compiled. I.e., these are the encryptions for which there does not exist some $y \neq 0$ for which $x * y = 0$, by checking that $\mathcal{E}(x) \boxtimes \mathcal{E}(y) \neq \mathcal{E}(0)$ for any encryption of 0. These are all the odd numbers (modulo 2^{32}, of which there are 2^{31} in total). Then by Lagrange's theorem, $x^{2^{31}} = 1$ in the modulus, so multiplying $\mathcal{E}(x)$ by itself 2^{31} times in the altered ALU gives an encryption of 1, for any x that is not a divisor of zero. That means feeding $\mathcal{E}(x)$ through $\mathcal{E}(x) \mapsto \mathcal{E}(x) \boxtimes \mathcal{E}(x)$ just exactly 31 times.

AB encryption

There are many more such arithmetic relations that an attacker may make use of in order to compute the encryptions of known constants. Rather than arguing for or against the efficacy of attacks that leverage these relations, we may attempt to make it impossible to create any 'constant by formula' by designing the altered ALU in a particular way. We may let the altered ALU encode nonsense for the result of x op x for any value x and any operation op. Then *any* attempt to calculate the encryption of 1 by means of x/x, or 0 by means of $x - x$, fails. So how may one calculate $x - x$ when one really wants to? Let there be *two* encryptions of each x, an 'A' and a 'B' encryption, thus: $\mathcal{E}^A(x)$ and $\mathcal{E}^B(x)$. Let same type encryptions 'A' and 'A' or 'B' and 'B' produce nonsense when sent through the altered ALU, but different type encryptions 'A' and 'B' or 'B' and 'A' produce the correct result. Thus $\mathcal{E}^A(x) \boxtimes \mathcal{E}^B(y) = \mathcal{E}^A(x * y)$ but $\mathcal{E}^A(x) \boxtimes \mathcal{E}^A(y)$ produces nonsense. It is easy to ensure – by compiling it in the right way – that a compiled program running in the KPU always puts validy typed operands through the ALU.

We have arbitrarily elected to let 'A' be the encryption type produced from an A- and a B-type encryption as operands, in that order, so we (symmetrically) design 'B' to be the encryption type produced from a B- and an A-type encryption as operands, in that order. If an attacker chooses an arbitrary encrypted value $\mathcal{E}^A(x)$, say, then passing it through the altered ALU's division operator produces nonsense from $\mathcal{E}^A(x) \boxminus \mathcal{E}^A(x)$, not $\mathcal{E}^A(1)$ or $\mathcal{E}^B(1)$, as the attacker would have hoped.

So this defence against the 'x/x attack' works. The penalty is that the cipherspace needs to be twice as large as it otherwise would be, in order to contain both the A- and the B- encryptions, and the altered ALU then needs

to be four times as large as it otherwise would be, in order to contain all the results of operations on same- and different-encryption operand pairs. Those are not really burdensome drawbacks in practice.

The real problem with 'AB-encryption', as we will call it, is that it is easily defeated by an attacker who knows only that the scheme is in use, and no more. If a pair of values $\mathcal{E}^A(x)$ and $\mathcal{E}^B(y)$ are observed by an attacker being passed through the ALU as operands to a single operation, then the attacker can be sure that these are indeed different type encryptions, or else the result of the observed operation would be nonsense. The attacker does not know which is the A- and which is the B-type operand, but he/she can be sure that putting them through the ALU one way round, then the other way round, will produce a result of different type each time. The expression $\mathcal{E}^A(x) \boxtimes \mathcal{E}^B(y)$ is of type A, and $\mathcal{E}^B(y) \boxtimes \mathcal{E}^A(x)$ is of type B. So one may legitimately divide the two expressions to get an encryption of 1:

$$(\mathcal{E}^A(x) \boxtimes \mathcal{E}^B(y)) \boxplus (\mathcal{E}^B(y) \boxtimes \mathcal{E}^A(x)) = \mathcal{E}^A(1)$$

Doing it the other way round yields an encryption in the other type:

$$(\mathcal{E}^B(y) \boxtimes \mathcal{E}^A(x)) \boxplus (\mathcal{E}^A(x) \boxtimes \mathcal{E}^B(y)) = \mathcal{E}^B(1)$$

Then one may calculate $\mathcal{E}^A(1) \boxplus \mathcal{E}^B(1) = \mathcal{E}^A(2)$ and $\mathcal{E}^B(1) \boxplus \mathcal{E}^A(1) = \mathcal{E}^B(2)$, and so on, to construct a codebook. Of course, the attacker does not have to be *certain* that he/she is observing a real bona fide calculation all the time – it is enough to be sure of finding an encryption of 1 *sometimes*, at some minimum rate, in order to be able to decrypt some statistically known fraction of all the encrypted computations observed.

ABC encryption

So AB-encryption does not foil an attacker, but the idea is not hopeless – the scheme needs refining. Instead, we propose *three* types of encryption: 'A', 'B' and 'C'. Then there are three distinct ways of encrypting a number x: $\mathcal{E}^A(x)$, $\mathcal{E}^B(x)$ and $\mathcal{E}^C(x)$, and we design the ALU so that operands of type 'A' and 'B' in that order give a result of type 'C', operands of type 'B' and 'C' in that order give a result of type 'A', and operands of type 'C' and 'A' in that order give a result of type 'B'. Any other inputs to the ALU give a nonsense result. In particular putting the operands in the wrong way round, as 'B' and 'A' in that order, for example, give a nonsense result.

The ABC scheme makes it impossible for the attack worked above with the AB scheme to succeed. We can show formally that there is no formula $a/b = 1$ conforming to ABC typing, where a/b is an expression like x/x containing no constants in which a and b are rearrangements of the same elements.

Lemma 1. *Under the ABC typing described above, there is no validly typed expression a whose elements can be rearranged to give a validly typed expression b of different type.*

Proof. Suppose for contradiction that there is a rearrangement of a as b such that a is of type A and b is of type B (say). Replace each operation op by multiplication '$*$' and each atom of type A by i, each atom of type B by j, each atom of type C by k, and get an expression $Q(a)$ that represents a as a multiplicative product in the quaternion system of '3-D integers' of the form $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$, where $x, y, z \in \mathcal{Z}$ are integers. The representation in quaternions follows the ABC typing: an A-type operand combined with a B-type operand should give a C-type result, and in quaternions that is correctly reflected as

$$\mathbf{i} * \mathbf{j} = \mathbf{k}$$

and likewise for $\mathbf{j} * \mathbf{k} = \mathbf{i}$ and $\mathbf{k} * \mathbf{i} = \mathbf{j}$, respectively reflecting that a B-type operand combined with a C-type operand gives an A-type result, and that a C-type operand combined with an A-type operand gives a B-type result. So we know that the representation $Q(a) = \mathbf{i}$ in quaternions, since a is of type A in the ABC scheme.

Representing b as a quaternion expression $Q(b)$ in the same way yields exactly the same product but for the fact that the elements appear in a different order and association in $Q(b)$ than in $Q(a)$. A different association makes no difference, since multiplication is associative in quaternions. Reversing the order of multiplication only changes sign, not direction, i.e.:

$$\mathbf{j} * \mathbf{i} = -\mathbf{i} * \mathbf{j} = -\mathbf{k}$$

and likewise for $\mathbf{k}*\mathbf{j} = -\mathbf{j}*\mathbf{k} = -\mathbf{i}$ and $\mathbf{i}*\mathbf{k} = -\mathbf{k}*\mathbf{i} = -\mathbf{j}$. So the representation for b is $Q(b) = \pm Q(a) = \pm\mathbf{i}$. But by hypothesis b is of type B, which means that $Q(b) = \mathbf{j}$ in quaternions. Since $\pm\mathbf{i} \neq \mathbf{j}$, the contradiction is proven. $\qquad\square$

The lemma does not require the same operations in the rearrangement, but of necessity there are the same number.

Remark. Examining the proof of Lemma 1 closely, while it does require the same variables and constants in the rearrangement, it does not require them to be repeated exactly the same number of times, but merely the same number of times up to parity: that is to say, if x appears an even (odd) number of times in a, then it must appear an even (odd) number of times in the rearrangement of a. Since the number of operations in the expression a is the same as in its rearrangement, the most that can happen by way of variation is to swap some two elements x of the original for two other variables y in the rearrangement. If $x/x - y/y = 0$ were typable in the ABC-scheme, that would be an example of such a swap.

That scotches the possibility of an attacker exploiting many possible 'big brothers' of $x/x = 1$. We believe that there is absolutely no possible 'big brother' at all, as expressed in the following formalization:

Conjecture 1. *There is no expression f that is validly typed in the ABC scheme, contains no constants itself, yet is equal in value to a known constant k.*

We cannot yet prove it completely, but we can come close (below).

Something like $x/x - x/x = 0$ would be a candidate for f, if only it could be typed in the ABC scheme. We conjecture that not only is this f impossible to type in the ABC scheme, but also any other expression f that gives a constant. That means that there is no formula that an attacker could apply 'blindly' using encrypted values and operations that have already been observed but whose unencrypted values are unknown in order to obtain the encryption of a known constant value. We should expect a proof to first show that $f = a/b$ or $a + b$ where a and b contain the same variables the same parity number of times. The lemma, as strengthened by the remark following it, then does the rest. Here is an intermediate result that obtains that first step via a very tiny extra hypothesis:

Proposition 1. *There is no expression f in which every variable appears an even number of times that is validly typed in the ABC scheme, contains no constants itself, yet is equal in value to a known constant k.*

Proof. Suppose $f = a/b$ or $f = a + b$, etc. Every variable appears with the same parity in a and b by hypothesis, which obtains the result via the remark following Lemma 1. $\qquad\square$

That leaves only the case when some variable x occurs an odd number of times in f to consider, in order to prove the conjecture fully.

We concentrate on the ABC scheme as a means of preventing attack on a homomorphic encryption via an algebraic formula. The next section constructs some concrete ABC schemes, simultaneously fitting two or more encrypted interpretations of arithmetic inside the same altered ALU for use inside a KPU.

3. ABC construction

If we look at a single bit then the arithmetic tables within the altered ALU of a KPU are as follows, where '$*$' signifies that any value is allowed, because the operation contemplated falls outside the ABC scheme:

$\boxplus$	$\mathcal{E}^A(0)$	$\mathcal{E}^A(1)$	$\mathcal{E}^B(0)$	$\mathcal{E}^B(1)$	$\mathcal{E}^C(0)$	$\mathcal{E}^C(1)$
$\mathcal{E}^A(0)$	$*$	$*$	$\mathcal{E}^C(0)$	$\mathcal{E}^C(1)$	$*$	$*$
$\mathcal{E}^A(1)$	$*$	$*$	$\mathcal{E}^C(1)$	$\mathcal{E}^C(0)$	$*$	$*$
$\mathcal{E}^B(0)$	$*$	$*$	$*$	$*$	$\mathcal{E}^A(0)$	$\mathcal{E}^A(1)$
$\mathcal{E}^B(1)$	$*$	$*$	$*$	$*$	$\mathcal{E}^A(1)$	$\mathcal{E}^A(0)$
$\mathcal{E}^C(0)$	$\mathcal{E}^B(0)$	$\mathcal{E}^B(1)$	$*$	$*$	$*$	$*$
$\mathcal{E}^C(1)$	$\mathcal{E}^B(1)$	$\mathcal{E}^B(0)$	$*$	$*$	$*$	$*$

$\boxtimes$	$\mathcal{E}^A(0)$	$\mathcal{E}^A(1)$	$\mathcal{E}^B(0)$	$\mathcal{E}^B(1)$	$\mathcal{E}^C(0)$	$\mathcal{E}^C(1)$
$\mathcal{E}^A(0)$	$*$	$*$	$\mathcal{E}^C(0)$	$\mathcal{E}^C(0)$	$*$	$*$
$\mathcal{E}^A(1)$	$*$	$*$	$\mathcal{E}^C(0)$	$\mathcal{E}^C(1)$	$*$	$*$
$\mathcal{E}^B(0)$	$*$	$*$	$*$	$*$	$\mathcal{E}^A(0)$	$\mathcal{E}^A(0)$
$\mathcal{E}^B(1)$	$*$	$*$	$*$	$*$	$\mathcal{E}^A(0)$	$\mathcal{E}^A(1)$
$\mathcal{E}^C(0)$	$\mathcal{E}^B(0)$	$\mathcal{E}^B(0)$	$*$	$*$	$*$	$*$
$\mathcal{E}^C(1)$	$\mathcal{E}^B(0)$	$\mathcal{E}^B(1)$	$*$	$*$	$*$	$*$

If, for example, we encrypt 0 and 1 as the numbers 1 and 2 in cipherspace under the A encryption, and as 3 and 4 under the B encryption, and 5 and 6 under the C encryption, then the tables are:

⊞	1	2	3	4	5	6
1	*	*	5	6	*	*
2	*	*	6	5	*	*
3	*	*	*	*	1	2
4	*	*	*	*	2	1
5	3	4	*	*	*	*
6	4	3	*	*	*	*

⊠	1	2	3	4	5	6
1	*	*	5	5	*	*
2	*	*	5	6	*	*
3	*	*	*	*	1	1
4	*	*	*	*	1	2
5	3	3	*	*	*	*
6	3	4	*	*	*	*

There is no instance, for example, of $x \boxplus y = x$ in these tables, amongst the values filled in.. Thus an attacker looking for an instance of $x \boxplus y = x$ in order to identify an encryption y of 0 will fail – unless the starred entries, which may be set to anything, inadvertently direct the attacker to it. If for example, $1 \boxplus 1 = 1$ is set, then 1 may be taken fortuitously by an attacker to be an encryption of 0, even though there is no significance in that result as far as the setter of the table is concerned.

What would prevent the table setter inadvertently implementing some equation that an attacker may check? There are two approaches: (a) to provide equally many indications that one element is the encryption of 0, or 1, as there are that another element is that encryption; (b) to provide no indications at all that point towards anything for any element.

Approach (b) is easy to implement. We want to avoid accidentally embedding *any* image of modulo-2 arithmetic in the tables. We can do that by avoiding any pair of elements x, y, such that the set of sums $x \boxplus x$, $x \boxplus y$, $y \boxplus x$, $y \boxplus y$ runs over the pair x, y again. Similarly for multiplication. We can specify the elements '*' in the table in order to make that so:

⊞	1	2	3	4	5	6
1	[3−6]	[3−6]	5	6	*	*
2	[3−6]	[3−6]	6	5	*	*
3	*	*	[5−2]	[5−2]	1	2
4	*	*	[5−2]	[5−2]	2	1
5	3	4	*	*	[1−4]	[1−4]
6	4	3	*	*	[1−4]	[1−4]

⊠	1	2	3	4	5	6
1	[3−6]	[3−6]	5	5	*	*
2	[3−6]	[3−6]	5	6	*	*
3	*	*	[5−2]	[5−2]	1	1
4	*	*	[5−2]	[5−2]	1	2
5	3	3	*	*	[1−4]	[1−4]
6	3	4	*	*	[1−4]	[1−4]

The { 2, 3} pairing is already sabotaged by $2 \boxplus 3 = 6$, and similarly for the other pairings off the diagonal. No attacker now has an excuse for thinking on the basis of an accidental equality $x \boxplus x = x$ that x is an encryption of 0, for example.

But what if an attacker is aware that we are using an ABC-scheme to hide the encryption? There is an approach that can confound an attacker in those circumstances – it is to make use of the freedom still available in constructing the tables in order to embed a different encryption, also under an ABC typing scheme of its own. The shape of the tables we already have gives the lead. The shape of the tables is, taking the entries two at a time:

⊞	A	B	C
A	*	C	*
B	*	*	A
C	B	*	*

⊠	A	B	C
A	*	C	*
B	*	*	A
C	B	*	*

and that has rendered only the types of the encryption. That ABC-scheme encryption occupies only three out of the nine slots available in each of these reduced tables, or three out of the six off-diagonal slots, if we grant that the diagonal slots are going to be constrained by the need to avoid inadvertently giving a hint to a naive attacker. So we can fit another ABC-scheme encryption into the remaining three off-diagonal slots like this:

⊞		A_2	C_2	B_2
		A_1	B_1	C_1
A_2	A_1	*	C_1	C_2
C_2	B_1	B_2	*	A_1
B_2	C_1	B_1	A_2	*

⊠		A_2	C_2	B_2
		A_1	B_1	C_1
A_2	A_1	*	C_1	C_2
C_2	B_1	B_2	*	A_1
B_2	C_1	B_1	A_2	*

This scheme only swaps the overall B- and C-type of the encryptions, but leaves room for the details of the encryption to be changed arbitrarily. We can set the new A-type encryption of 0, 1 to be the opposite of the old, for example: 2 for 0 and 1 for 1. We may set the new B-type encryption to be the same as the old C-type encryption: 5 for 0 and 6 for 1. Then we can set the new C-type encryption to be the opposite of the old B-type encryption: 4 for 0 and 3 for 1. That fills out the tables as follows:

⊞	1	2	3	4	5	6
1	[3−6]	[3−6]	5	6	3	4
2	[3−6]	[3−6]	6	5	4	3
3	5	6	[5−2]	[5−2]	1	2
4	6	5	[5−2]	[5−2]	2	1
5	3	4	1	2	[1−4]	[1−4]
6	4	3	2	1	[1−4]	[1−4]

⊠	1	2	3	4	5	6
1	[3−6]	[3−6]	5	5	4	3
2	[3−6]	[3−6]	5	6	4	4
3	6	5	[5−2]	[5−2]	1	1
4	5	5	[5−2]	[5−2]	1	2
5	3	3	2	2	[1−4]	[1−4]
6	3	4	2	1	[1−4]	[1−4]

There are 8 ways of filling out the off-diagonal elements of the table, each corresponding to one way of encrypting 0, 1 in each of the new A, B and C types. There is a free choice of which member of the designated pair of cipherspace elements is 0, and which is 1, for each of the A, B and C pairs. Each 'way' gives a coding of 0, 1 in each of the A, B, C types. One of these ways is exactly the same as the old coding prefilled in the table – that is 1, 2 for 0, 1 in the new A-type encryption; 5, 6 for 0, 1 in the new B-type encryption; 3, 4 for 0, 1 in the new C type encryption. The rest give rise to encodings that are different from the original, which means that at least one of 0 or 1 is coded differently from the original in at least one of the A, B, C types. Given an ALU implementing these 1-bit arithmetic tables, an attacker cannot tell which of two different encodings is in use.

However, an attacker might be able to observe computations actually taking place, which should mean that which subset of the tables is actually used can be determined, and along with it which coding is in use. Therefore we phrase the first open question:

Question 1. *Is there an embedding of two different ABC encryptions for 1-bit arithmetic into the same 6×6 tables such that the embeddings overlap?*

An ALU implementing overlapping encryptions would be far harder to crack than the non-overlapping encryptions shown above. But so far as we can determine, the answer is 'no'. Our searches have failed to produce any example.

Question 2. *How many simultaneous embeddings of two different ABC encryptions for 1-bit arithmetic into the same 6×6 tables can be managed? How many overlapping ones?*

We believe the answer is 2 non-overlapping embeddings, and 1 overlapping (i.e. overlapping cannot happen).
What about embedding in larger cipherspaces? The cipherspace here is size 6, leading to arithmetic tables of size 6×6. If we raise the size to 7, then there will be 1 non-coding value in the cipherspace. That is, it will have no significance as an A, B- or C-type encryption. We will call it 'X-type'. An X-type operand put through any operation with any other operand is allowed to take any value at all.

Question 3. *Does embedding in a larger cipherspace, containing additional 'X-type' padding values that may be combined by the arithmetic tables to give any value at all, make any difference to the answer to Questions 1 and 2?*

Although 1-bit arithmetic is sufficient, in that black-box hardware units implementing 1-bit arithmetic can be combined to produce 8-, 16-, 32-bit arithmetics as required, and each bit in the assemblage can make use of a different fundamental encryption, one may ask if larger fundamental units give any advantage to a defender. 2-bit arithmetic, for example, encodes 4 numbers, 0, 1, 2, 3, in A-, B- and C-type encryptions, requiring four 12×12 tables, one each for addition, subtraction, multiplication and division. The construction given in this section shows that one may simultaneously embed two different ABC encryptions at the same time, but what is the limit? We believe that the answer is "at least four" for 2-bit arithmetic in 12×12 tables, but what about 1-trit arithmetic (modulo 3 arithmetic) in 9×9 tables?

Question 4. *How many different ABC encryptions of modulo-n arithmetic may simultaneously be embedded into size* $3n \times 3n$ *tables? May the embeddings overlap? What if the cipherspace is extended to contain m non-coding values, so the tables are of size* $(3n + m) \times (3n + m)$*?*

One significant sizing is that which accommodates 10-bit arithmetic (i.e., modulo-n arithmetic, for $n = 2^{10}$) embedded in a 12-bit cipherspace (i.e., with $m = 2^{10}$ non-coding values). The reason that it is significant is that it is the smallest size for which one may use a 12-bit AES [8] encryption algorithm (with a 12-bit key on a state consisting of a 4-dimensional vector of 3-bit ordinates from a finite field of size 8) to construct the tables, thus leveraging known cryptographic theory. Decryption means applying the key to a 12-bit 'cipherspace' number to get a 12-bit 'plaintext' number, and dividing by 4 to get the 10-bit arithmetic number it represents. The remainder modulo 4 gives the type – 0 for 'A', 1 for 'B', 2 for 'C', and 3 for 'X' (not a type). To construct the table for this embedding, decrypt the operands, and if they are of type 'A' and 'B', or 'B' and 'C', or 'C' and 'A', then compute the result of the arithmetic operation modulo 2^{10}, multiply by 4 and add 0, 1 or 2 according as the result type is 'A', 'B' or 'C' respectively according to the ABC rules. If the operands do not match this pattern, then one can encode any result value in the table. Employing an AES algorithm makes it possible to implement an altered ALU in hardware by surrounding an ordinary 10-bit ALU by 12-bit AES codecs, since AES encryption is fast in hardware.

But simply programming a PROM with the tables is also possible, because 12-bit cipherspace means each of the four arithmetic tables takes up 12×2^{24} bits, or less than 1MB. That is practical today.

Finally we may ask for computational searches of valid expressions in ABC-typing that give rise to a constant result, falsifying Conjecture 1.

Question 5. *Are there any expressions in addition, multiplication modulo 2, containing only variables x, y, of types A, B respectively, and no constants, subject to the rules of ABC typing (that AB=C, BC=A, CA=B) in which x appears an odd number of times, and y an even number of times, which is equal to a constant value?*

That formalizes a search that cannot prove Conjecture 1, but could disprove it, or give valuable insights. The restrictions arise because one may reduce an expression in multiple variables to an expression in just three variables, one of type A, one of type B, one of type C, by substituting all variables of type A by just one of the variables of type A, etc. Then one may replace a z of type C by xy, leaving just two variables. We do not have the computational resources with which to investigate the question via brute force search, but a result like 'there is no expression in less than 100 operations with the properties sought' would be very convincing.

4. Summary

This paper has described ABC encryption, and explained why it is thought to be significant in the context of the security of all-encrypted processing. Attacks on a crypo-processor (KPU) can be thwarted by such an encryption. It has been proved that ABC encryption prevents an attacked from using a formulaic approach based on observations of arithmetic performed by the KPU. We have set out several important questions, which we hope will raise interest among researchers with the resources to conduct brute force searches to see how robust the approach propose din this paper is in practice.

References

[1] P. T. Breuer, J. P. Bowen, A fully homomorphic crypto-processor design: Correctness of a secret computer, in: Proc. 5th Intl. Symposium on Engineering Secure Software and Systems, no. 7781 in LNCS, Springer, 2013, pp. 123–138.

[2] F. Bouesse, M. Renaudin, A. Witon, F. Germain, A clock-less low-voltage AES crypto-processor, in: Proc. 31st European Solid-State Circuits Conf. (ESSCIRC) 2005, 2005, pp. 403–406. `doi:10.1109/ESSCIR.2005.1541645`.

[3] R. Buchty, N. Heintze, D. Oliva, Cryptonite – a programmable crypto processor architecture for high-bandwidth applications, in: Organic and Pervasive Computing – ARCS 2004, no. 2981 in LNCS, Springer, 2004, pp. 184–198. `doi:10.1007/978-3-540-24714-2\_15`.

[4] H. W. Kim, S. Lee, Design and implementation of a private and public key crypto processor and its application to a security system, IEEE Transactions on Consumer Electronics 50(1) (2004) 214–224. `doi:10.1109/TCE.2004.1277865`.

[5] D. A. Patterson, Reduced instruction set computers, Communications of the ACM 28(1) (1985) 8–21.

[6] P. T. Breuer, J. P. Bowen, Typed assembler for a RISC crypto-processor, in: Proc. 4th Intl. Symposium on Engineering Secure Software and Systems, no. 7159 in LNCS, Springer, 2012, pp. 22–29.

[7] C. Gentry, Fully homomorphic encryption using ideal lattices, in: Proc. 41st ACM Symposium on Theory of Computing, ACM, 2009, pp. 169–178. `doi:10.1145/1536414.1536440`.

[8] V. R. Joan Daemen, The Design of Rijndael: AES – The Advanced Encryption Standard, Springer, 2002.