Idea: Towards a Working Fully Homomorphic Crypto-processor

Practice and the Secret Computer

Peter T. Breuer[1] and Jonathan P. Bowen[2,⋆]

[1] Department of Computer Science, University of Birmingham, UK
ptb@cs.bham.ac.uk
[2] School of Computing, Telecommunications and Networks,
Birmingham City University, UK
jonathan.bowen@bcu.ac.uk

Abstract. A KPU is a replacement for a standard RISC processor that natively runs encrypted machine code on encrypted data in registers and memory – a 'general-purpose crypto-processor', in other words. It works because the processor's arithmetic is customised to make the chosen encryption into a mathematical homomorphism, resulting in what is called a 'fully-homomorphic encryption' design. This paper discusses the problems and solutions associated with implementing a KPU in hardware.

1 Introduction

A KPU ('**K**rypto-**P**rocessor **U**nit') is a simple general purpose processor and processor architecture that works on data in encrypted form. To input data into the machine, the owner prepares it in encrypted form and receives encrypted data back. In theory, a KPU need never decrypt, even as it places encrypted data in memory and registers and runs the encrypted program. That makes it of interest for cloud computation, and also the reverse situation, where, for example, a bank wishes to securely devolve responsibility for transactions on individual bank accounts to personal chips held by the untrusted account owners.

The mathematics relates a KPU to the science of fully-homomorphic encryption, first introduced as privacy homomorphisms in [10]. Well-known encryptions such as RSA private/public key cryptography [11] exhibit partial homomorphism, in the case of RSA with respect to multiplication, in that $RSA(a)*RSA(b)$ mod $m = RSA(a*b)$, where RSA stands for the encryption and m is its associated (large) arithmetic modulus. An encryption that supports homomorphism both with respect to addition and to multiplication is said to be a fully homomorphic encryption (FHE). The operations on encrypted data corresponding to multiplication and addition on the unencrypted data need not be as simple as multiplication or addition in the general case, but Gentry [6] constructed a FHE in which those operations, while complex, do not compromise the encryption

⋆ Jonathan Bowen acknowledges the support of Museophile Limited.

J. Jürjens, F. Piessens, and N. Bielova (Eds.): ESSoS 2014, LNCS 8364, pp. 131–140, 2014.
© Springer International Publishing Switzerland 2014

and thus may still be carried out by untrusted parties. In a KPU, the corresponding operations are embedded in the hardware. A KPU can be handed out to an untrusted party just as the software algorithms for the operations that work on FHE-encrypted data can be handed out. However, a KPU design allows more flexibility in the choice of the underlying encryption, and engineering tradeoffs come to the fore. One may entertain, for example, the possibility of operations that would reveal the encryption if they were exposed, but which are implemented in hardware that physically secures them, as for SmartCards [8].

In terms of its construction, a KPU is a processor in which the standard arithmetic logic unit (ALU) has been replaced by a different design satisfying certain special properties. A standard processor is the trivial case. A KPU is described in mathematical terms in [1], where it is shown that, whatever the detail of the implementation, provided the modified ALU satisfies the required properties then a KPU operates *correctly*, in that the machine states that obtain during the execution of an encrypted program are encryptions of the states that would result in an ordinary RISC [9] processor running the unencrypted program. A RISC design is a convenient point of departure for the proof, because of its simplicity, but it is equally possible to build a KPU by starting from another class of modern von Neumann processor.

The modified ALU in a KPU, instead of $1 + 1 = 2$, does $6769875\#6769875 = 87997001$ (for example), those numbers being encodings of 1 and 2 under the encryption. In its most general form, this is a homomorphism statement, and the 'special properties' of the modified ALU alluded to above are the requirements that its operations, functions and relations be homomorphic images of the standard operations.

Counter-intuitively for those who appreciate that the slightest change inside the processor may snowball, the grossly changed arithmetic results in states that are 'correct', but encrypted. One can liken it to changing from speaking 'English' in a CPU to speaking 'Chinese' in the KPU, with the added difficulty of very many 'Chinese' words for every 'English' word. The detail in a practical KPU design is merely aimed at avoiding design features that may inadvertently sabotage this principle. It is important, for example, to separate the circuitry that does arithmetic on program addresses from that which does arithmetic on data, or the encrypted '+1' on the address at most every tick of the clock would leak significant information, as well as compromise program loading and caching localities. In consequence, while data addresses and contents and program instructions are encrypted in a KPU, program addresses should be encrypted differently and may not be encrypted at all. Running programs must be written to keep the two kinds apart, so that encrypted values are acted on by instructions that expect encryption, and unencrypted values are acted on by instructions that do not expect encryption [2].

The idea behind KPU design is 'problem reduction'. It reduces the problem of encrypted general purpose computing to the lower order problem of constructing an appropriately modified ALU. In principle a very large lookup table suffices to drive the ALU, but replacing every gate in the standard ALU with an 'encrypted

version' of the same gate also works. Between those extremes lie many other possibilities, which will be explored in this paper.

Speed is not a primary concern — IBM's FHE implementations take on the order of seconds per single bit operation on a vector mainframe [7], though this shows signs of being improved by means of special techniques on GPU-based hardware [12] — and there are many other factors to consider. There is, for example, a *hardware aliasing problem*, which arises because ciphers are one–many and thus many different ('Chinese') encryptions of a single ('English') memory address may crop up and be used during the execution of a program. Since all the different aliases designate physically different locations in memory, a working program for a KPU must be written to access only one of them, which means every address must be calculated the same way at every use [3].

This paper focuses on two areas in working KPU design in particular. Section 2 discusses hardware options and Section 3 discusses encoding strategies.

2 Word Size and Hardware Design

The first issue in processor design is 'how big is a data word', the physical extent of the standard unit of data. 'Large' means the processor needs long registers and many traces and wide busses to carry the data internally, which is costly. The quick answer here is that nobody yet really knows what word size is best, because different design approaches indicate different solutions.

In a KPU, the data word size is the encryption block size, the size of a unit of encrypted data. For strong encryption, about 128 bits is reasonable for many of the common ciphers in the medium to long term, whereas 64 bits is borderline, but sufficient for real time protection, supposing key size the same as encryption block size. Neither is technically impossible, but 64 bits would be very favourable from the manufacturer's point of view as the associated technology is already in use. Fewer bits would be even better from that perspective, however. The trade-off is small size (low cost, low power) against greater security.

How many plaintext bits does a 64-bit encrypted data word contain? It can be anywhere from 1 to close to 64, leaving room for padding bits that make the encryption one–many overall; 32 plaintext bits and 32 padding bits would result in 2^{32} different encodings of each 32-bit plaintext number. The numbers are significant because if an attacker guesses the encryption for 1 and also guesses which operation is the '+' in the ALU, then, given access, the attacker can generate the encryption of 2 via $1 + 1$, of 3 via $2 + 1$, etc, and thus obtain a complete codebook. Many encodings for each plaintext number imply many codebooks and only relatively few encountered in any program run. That is reassuring because, in general, there is no mathematical analysis available of the security of arithmetic in a KPU. That is also the case for white-box access, but note that the example of the encrypted arithmetic in Gentry's software solution [5] shows that it is not a priori unsafe to permit unfettered access to hardware. To make the discussion concrete, five designs are set out below.

1. Embedded codecs: We may create an encrypted ALU by placing codecs on inputs and outputs of a standard ALU, as in Fig. 1. The codecs (D, C) contain

keys that must be transferred securely into the hardware, perhaps via a Diffie-Hellman protocol [4], and they must be invulnerable to electronic probes. That is within the capabilities of SmartCard manufacturers today. The key should be volatile, so it does not survive disconnection, and the hardware's internal traces protected by overlying circuit elements. A 64-bit (encrypted) word is inherently feasible, but stripping out and replacing 32 bits of padding requires extra hardware. The simplest implementation has the data bits in the middle of the word and routes them to a 32-bit ALU. The output padding is generated by a separate unit (P); it can multiply input paddings and take the middle 32 bits ('mid-out' hash), folding in extra randomness as desired.

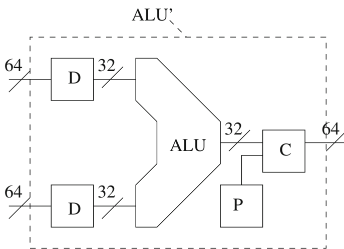

Fig. 1. Building an encrypted ALU using codecs (D, C) with embedded keys, and a padding unit (P). 64-bit inputs at left, 64-bit outputs at right.

One problem with this kind of design, apart from potential vulnerabilities with harbouring keys, is that the speed of the codecs limits the speed of the processor. Some encryptions when done in hardware can run at a few hundred MHz (CuBox run an ARM v6-based chip doing AES encryption clocked at 800MHz).

Nevertheless, the design is easily realised with present-day silicon technologies, requiring no radical innovations, and represents the most likely initial implementation technique. A small company with processor expertise can already produce chip wafers based on this idea.

2. Lookup tables: Fig. 2 shows a 16-bit 'black-box' ALU design. It contains tables for encrypted arithmetic on two 16-bit encrypted inputs, producing one 16-bit encrypted output. Leaving security questions aside (16-bit cipherspaces are easily searched, but that is not the end of the story), each binary operator requires tables occupying $(2^{16})^2 \times 16 = 2^{36}$ bits, or 8GB. That is too large to go in-processor at present, but it can reside in RAM. Every arithmetic operation must access the tables, which limits speeds to 200MHz to 400MHz, in practice. But we expect advances in technology to make the numbers feasible in a few years, returning focus to the security question here.

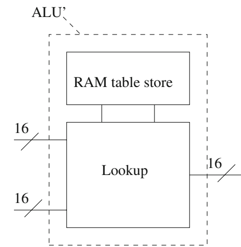

Fig. 2. Building an encrypted ALU as a 'black-box' lookup table requires large amounts of on-board RAM. 16-bit inputs at left, 16-bit outputs at right.

This solution focuses on encrypted arithmetic tables as the encryption 'key'. Those 8GB lumps of data in RAM need to be supplied, probably over the Internet, at relatively frequent intervals as different configurations are adopted for different encryptions, but sending them beforehand, or in parallel while computation proceeds, is an option. The transfer need not be in public view, but if it

is, then can an attacker work out an encoding given the full tables, together with observations of what computations are done in practice?

The answer is formally 'no' (one may place two and in practice 'very many' encodings simultaneously in the same tables; see Section 3), but observations of a running CPU may aid the attacker. If this solution is secure, then it can also be implemented in software, leading to the safe running of a KPU in simulation.

The problematic aspect of this solution is the short 16-bit input and output sizes. If, say, 8 bits of that is padding, it only leaves room for 8 bits of data beneath the encryption. While 8-bit computation is feasible (and from 4 to 16 times as slow as 32-bit computation), it is disadvantaged in security terms because 32-bit calculations take several cycles, and the carry in to the second cycle will be highly constrained and yet encrypted in the same way as the other inputs, which makes decoding relatively easy.

On the plus side, however, is that algebraic attacks using the ring structure of addition and multiplication under the homomorphism do not work. One might look for, say, 3 as one of the highest-order elements of the tables under self-multiplication (it should take or 2^{16} or 16 self-multiplications to get back to 3 again, depending how one counts), but that approach is scotched because the padding makes the result still look different from the original.

3. Modular design: Putting several ALUs in parallel in the hardware allows the lookup table solution to be 'ramped up' to deliver 32-bit computation in one cycle in hardware, as shown in Fig. 3 for addition. The individual encrypted adders are 16-bit for a total of 64 bits of input and output, but the encryption is different in each group of 16 bits. The encryption varies again on each of the carry outputs and inputs. One may imagine that between each adder an arbitrary extra encryption has been applied via codecs D_i, C_i, shown in dotted lines, but in reality these are folded into the tables.

In hardware, no internal connections are exposed, but this solution is problematic in software. Can the units in solid lines be safely stored as lookup tables in full public view? The answer is formally unknown.

One may embed at least two different ciphers in one 3-bit encrypted arithmetic table, encoding just one data bit (the technique is explained in Section 3). The maximum number M of ciphers is much higher than two, but there are too many configurations to establish M exactly (a 3-bit table for one arithmetic operation has 64 entries, each 3 bits, thus $8^{64} = 2^{192}$ tables to explore). The significance of that may be seen via an analogy: Suppose that the English word 'mouse' is also the Chinese word for 'sunshine', with similar overlaps for all English and Chinese words. The situation here is then that an observer cannot decide if an observed computation is an 'English' conversation about pests or a 'Chinese' conversation about weather. The mathematics makes the 'grammar' (the arithmetic) look right both ways.

The layout of Fig. 3 may be adapted for 32 3-bit units, each encoding one bit of data. Then 32-bit computation is implemented with 96-bit encrypted words and an attacker with full access must explore M^{32} valid decodings, assuming it is already known which decodings are valid for those tables (there are only

$8!/2! = 20160$ each to check in the 3-bit case). The tables are small and may be placed in-processor. Alternatively, 21-bit computation (via 21 3-bit units in the layout of Fig. 3) can be fitted in 63-bit encrypted words, and 24-bit computation in 64-bit encrypted words is feasible using trits and base-6 digitisation.

The lemma to remember here is that the arithmetic tables for coded values do not expose the coding, when padding makes the coding 1-to-many.

We will elaborate the approach in §5 below. First consider another approach that at first also looks unlikely.

4. Gate-level encryption: One may replace every single (1-bit) trace in an ALU by a set of 3 or 8 or 16, etc., traces carrying respectively a 3- or 8- or 16-bit encryption of the single bit; every OR gate is replaced by a corresponding 'encrypted-OR gate', possibly table-driven. If we consider the units of Fig. 3, then each of them may be implemented via a network of such 'encrypted gates'.

This reduces the design problem to dealing with just one bit of data, encrypted in as many bits as may be advisable for security. Fig. 4 shows how a 16-bit encrypted one-bit half-adder may be implemented like this. Instead of one table for addition and another for multiplication, etc., there is one table for 1-bit AND, one for 1-bit OR, and one for 1-bit inversion, but, in theory, just one table, for 1-bit NAND, will suffice. So storage requirements are 'only' one 8GB table for each 16-bit encryption used.

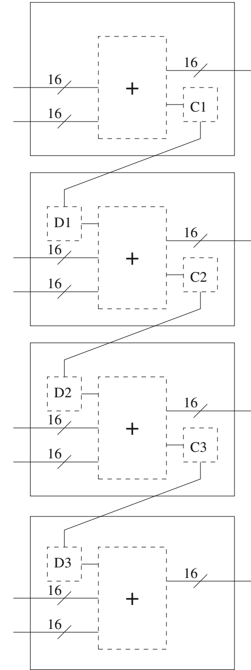

Fig. 3. Encrypted adder built from smaller units, with 4×16-bit inputs at left, 4×16-bit output at right, and distinct encodings in each unit.

An entire 16-bit encrypted ALU can be built in this way, using just one encryption, but it requires 16×16 input and output traces, i.e., encrypted words of 256 bits. But there is a problem: how to access the arithmetic tables simultaneously for all the gates that need to. In practice, with today's technology, one cannot. Even so, ALUs tend to be built so that calculation propagates across them in systolic fashion. If the construction is at worst n gates wide by m deep (m determines the latency), then the calculation may in principle be pipelined using just n gates and n tables organised into m stages. Each stage of the calculation takes time equal to one table lookup, thus a complete arithmetic calculation should take time equal to m table lookups. But one complete calculation will exit the pipeline at intervals of one lookup, so the throughput is normal. Pipelined ALUs (for floating point arithmetic) are common in processor technology.

5. Hybrids: We now revisit the modular design of Fig. 3. That has relatively weak 4×16-bit security in the configuration shown, but imagine an AES codec that *decrypts* a 64-bit ciphertext to a 64-bit plaintext, but which does so in

hardware. AES works via addition and multiplication operations on 16-bit segments. Apply the process that built Fig. 4 to the codec, operation by operation.

Imagine the ith 16-bit segment of plaintext output as followed by a new encoder E_i that produces 16 bits of encrypted output. Pass E_i to the input side of the adjacent internal AES operation G, replacing it with an encrypted operation G', such that $E_i \cdot G' = G \cdot E_i$. Repeat, passing the encoders E_i from output to input side of each successive layer of internal AES operations in turn.

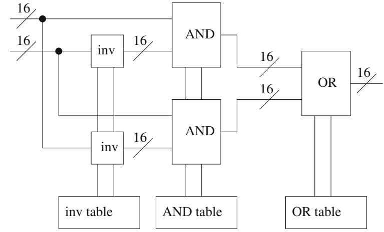

Fig. 4. An adder built from encrypted gates. 16-bit inputs at left, 16-bit output at right.

The process ends in a design analogous to Fig. 4, shown at left in Fig. 5 as 'D'. In it, every 16-bit operation G in the original AES decoder has been replaced by an 'encrypted version' G' working on 16-bit encrypted words. Internal changes between encryptions E_i and E_j are notionally handled by extra codecs Di and Cj shown in Fig. 5, although in reality these are folded into the tables that drive the different encrypted operations. What does this bizarre construction do?

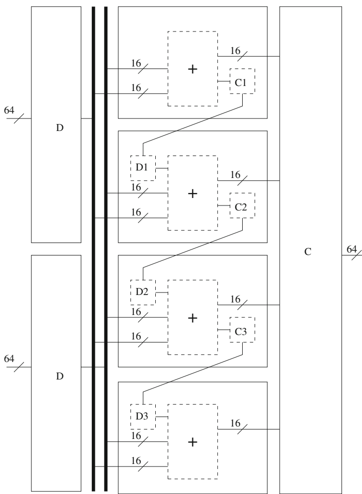

Fig. 5. Increasing encryption security. 64-bit inputs at left, 64-bit output at right.

The answer is that it takes as input $4 \times 16 = 64$ bits in which the ith group of 16 is an encoding under E_i of the ith 16-bit segment of the AES encoding of a 64-bit plaintext number x. The output is a 64-bit word y in which the ith group of 16 bits is the encoding under E_i of the ith 16-bit segment of x. In other words, it decodes doubly encrypted data to singly encrypted data that is suitable as input to the design of Fig. 3. The AES key is kept encrypted under the E_i in the internals of the modified decoder.

After decoding by D, doubly encrypted data is suitable for handling by the encrypted arithmetic structure of Fig. 3, shown in the centre of Fig. 5. The output from that may be recoded again using a modified AES encoder, labelled C in Fig. 5, producing doubly encrypted data. The AES keys are stored encrypted, and the ALU of Fig. 5 does 'doubly encrypted arithmetic'. An untrusted party with the encrypted AES keys can decrypt doubly encrypted data to singly encrypted data, but no more.

What is the advantage of this construction? The input encoding is at least as safe as AES. The keys, even if uncovered, are themselves encrypted and do not serve to decrypt the input, or output, or any intermediate. But the number of

bits in the construction is most significant: it is just 64 bits. If every trace and gate had been 'encrypted', as in §4, it would have been 32×64 bits.

So what is the right word size? Every size from 3, 8, 16, 64, 256, 512 bits has been suggested above. It requires detailed simulation and negotiation with chip manufacturers to make the decision in practice.

3 ABC Encoding

If the KPU's ALU contains a division operator – or even if there is a division routine in software – then an attacker with sufficient access can nearly always obtain an encryption of 1 by computing x/x for any encrypted x that has been observed. Even if which operation is division is formally unknown, the choice of n (usually 64 at most in a conventional ALU) operations merely multiplies up the number of possibilities to be tried by n, which is not significant. And obtaining encrypted 1 gives encrypted 2 via $1+1$, etc, until a complete codebook is constructed. This attack has implications for the encodings used in a KPU.

We have remarked that padding under the encryption is justified by the need for many codebooks in order to confound this kind of attack, but there is also a separate 'defense by construction' that may be built into the system. It is to set the ALU so that $x \,\mathrm{op}\, x$, always gives a nonsensical or random result, for any operator. How then to *really* calculate 1/1, for example? Two different and disjoint encodings are implemented, a type-A encoding and a type-B encoding, and $\mathrm{A\,op\,B}$ gives the right answer of type B, while $\mathrm{A\,op\,A}$ and $\mathrm{B\,op\,B}$ always give nonsense. Symmetrically, $\mathrm{B\,op\,A}$ gives the right answer of type A. It is simple to compile programs such that every operation takes place on operands of types A and B, or B and A, and the program runs correctly. This is called 'AB encoding'.

Unfortunately, AB encoding does not make things more difficult for an attacker with sufficient access. Doing the calculation $(\mathrm{x}+\mathrm{y})/(\mathrm{y}+\mathrm{x})$, where x is of type A and y of type B, still gives 1. An attacker may use any observed x, y.

An improvement called 'ABC encoding' resolves the problem. It adds one more disjoint encoding, a so-called type-C encoding, to the mix. The valid operations are now of type $\mathrm{A\,op\,B} = \mathrm{C}$, $\mathrm{B\,op\,C} = \mathrm{A}$ and $\mathrm{C\,op\,A} = \mathrm{B}$, and everything else gives nonsense. Again, compiling programs so that all operations have correct typing is trivial. One may prove that an attacker cannot take an observed calculation x of type A, reshuffle its parts to obtain a calculation y of type B, and then compute x/y with ABC encoding, for a known constant result. The logic of ABC encoding does not allow it. A loophole, however, is that the attacker may not merely reshuffle parts, but also duplicate or eliminate some parts in a revised sum, to get a constant. The following calculation is valid in ABC encoding:

$$(x * y) + (y * (x * y)) = 0 \mod 2$$

So an attacker with the encrypted arithmetic tables for two 1-bit plaintext operations can obtain an encoding of the 1-bit plaintext '0'. If the attacker does not know which operation is which, however, then there is no way of getting a constant result out (mod 2), and nothing can be gained in this way.

+	1	2	3	4	5	6
1	2	1	5	6	3	4
2	1	2	6	5	4	3
3	5	6	4	3	1	2
4	6	5	3	4	2	1
5	3	4	1	2	6	5
6	4	3	2	1	5	6

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

Fig. 6. Two different encodings simultaneously embedded in the same 6×6 tables for encrypted addition and multiplication mod 2, using ABC encoding. The 'lower left' A, B, C encryptions of 0,1 are 1,2;3,4;5,6 respectively. The 'upper right' encryptions are 2,1;6,5;3,4 respectively. Only AB=C, BC=A, CA=B gives valid results, such as A0+B1=C1, the rest are arbitrary.

ABC encoding trebles the size of the cipherspace required, and thus requires nine times as much storage space for arithmetic tables, as well as slightly increasing the number of bits required for an encrypted word. However, several different encryptions may be placed in the same tables simultaneously. Fig. 6 shows an example in the minimal possible size ABC tables: 6×6 in a cipherspace of size 6 for modulo 2 arithmetic.

In practice, 8 coding values would be used for 3 full bits of cipherspace, the redundancy permitting more overlaps. However, the upper limit for overlap as the cipherspace size increases is not known, nor is the number $M \geq 2$ of different encryptions that may be fitted in simultaneously. Despite the formal unknowns, we believe that ABC encodings do make it much more difficult to deduce the encryption from the encrypted arithmetic tables. An attacker may never hope to recognise a chance encoding that looks like $1 * 1 = 1$ under ABC rules, for example, because patterns of that form may never be constructed.

Fig. 6 proves that the tables do not determine the encryption uniquely.

4 Conclusion

The construction of the modified arithmetic logic unit in a KPU (a general purpose fully homomorphic crypto-processor architecture) has been discussed, with options ranging from monolithic lookup tables to gate-wise encryption. The objective is the implementation of PC-sized or smaller-sized computers that do all their work encrypted, with applications in many areas in the realm of secure computing. There is a close relation with work on fully-homomorphic computing, with hardware replacing the role of software algorithms. Some design options use

no or an incomplete set of keys – meaning that the processor itself does not know the encryption it uses – which implies that no backdoor can ever be built in.

'ABC encoding' has been introduced as a technique that we believe always improves security in a fully homomorphic context, potentiating the use of smaller encryption block-sizes.

References

[1] 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.) ESSoS 2013. LNCS, vol. 7781, pp. 123–138. Springer, Heidelberg (2013)
[2] Breuer, P.T., Bowen, J.P.: Typed Assembler for a RISC Crypto-Processor. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) ESSoS 2012. LNCS, vol. 7159, pp. 22–29. Springer, Heidelberg (2012)
[3] Breuer, P.T., Bowen, J.P.: Certifying Machine Code Safe from Hardware Aliasing: RISC is not necessarily risky. In: Counsell, S., Núñez, M. (eds.) Proc. OpenCert 2013, collocated with SEFM 2013. LNCS. Springer, Madrid (to appear 2013)
[4] Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976), doi:10.1109/TIT.1976.1055638.
[5] Gentry, C.: Computing arbitrary functions of encrypted data. Communications of the ACM 53(3), 97–105 (2010)
[6] Gentry, C.: Fully Homomorphic Encryption Using Ideal Lattices. In: Proc. 41st ACM Symposium on Theory of Computing (STOC), pp. 169–178. ACM (2009), doi:10.1145/1536414.1536440, ISBN: 978-1-60558-506-2
[7] Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011)
[8] Kömmerling, O., Kuhn, M.G.: Design principles for Tamper-Resistant Smartcard Processors. In: Smartcard 1999, Chicago, Illinois, USA, May 10-11, pp. 9–20 (1999)
[9] Patterson, D.A.: Reduced Instruction Set Computers. Communications of the ACM 28(10), 8–21 (1985)
[10] Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation 32(4), 169–180 (1978)
[11] Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
[12] Wei, W., et al.: Accelerating Fully Homomorphic Encryption on GPUs. In: Proc. IEEE High Performance Extreme Computing Conference (2012)