Superscalar Encrypted RISC
Measure of a Secret Computer for Encrypted Computing

Peter T. Breuer Jonathan P. Bowen Esther Palomar Zhiming Liu

Abstract—**Appropriately modifying the arithmetic embedded in a processor causes user data to remain in encrypted form throughout processing, providing a technical approach to privacy against the operator and other powerful insiders. The theory has now been checked in practice via a prototype superscalar processor design that fully implements a standard OpenRISC binary interface, modified for encrypted running. The processor runs in a (one-to-many) encryption in user mode, and runs unencrypted in operator mode. We report here cycle-accurate measurements on models that incorporate AES-128 (symmetric, keyed; the US 'advanced encryption standard') and Paillier-72 (asymmetric, additively homomorphic, no key in-processor) hardware-based encryption on a 128-bit platform, and RC2-64 encryption (symmetric, keyed) on 64-bit. With the symmetric encryptions, the models run at 104–140 MIPS on a 1 GHz base clock, measured on the Dhrystones 2.1 benchmark, equivalent to a 433–582 MHz classic Pentium. The practical experience gained allows us to propose a RISC instruction set modified for encrypted working under which semantic security for user data against the operator is mathematically guaranteed on this kind of platform, in cooperation with an 'obfuscating' compiler. The aim of this paper is to raise awareness in the light of recent events that computing provably secure against insiders has not only been logically achievable for years, but is also plausibly practical and marketable.**

1. Introduction

If the arithmetic in a conventional processor is modified appropriately, then the processor continues to operate correctly, but all its states are encrypted [1], which means that encrypted data is read and written at encrypted addresses, and both data and addresses pass through the internal registers of the machine in encrypted form. That is *encrypted computing*. Running an appropriate machine code instruction set, it turns out to be mathematically impossible for the operator to infer from the computation either statistically or logically what the user's encrypted data means, despite having read and write access to it and the program code [2]. That is to say, secure against-the-operator computing in the cloud on behalf of a remote user is 'formally solved' by the approach and it remains to be shown that a processor design following it is practical.

1.1 **Objectives.** This paper gives performance measures (our latest models achieve numbers equivalent to a fast classic Pentium) on a single pipeline processor design with the aim of challenging the hardware community to apply the approach to encrypted computing with more state-of-the-art computer architectures than our prototype's, and challenging the security community to get to grips with the ideas.

Another objective of this paper is to convince that it does work via the account here. That this class of processor has nothing but a changed arithmetic inside is hard on an engineer, while processors are not generally the domain of theoretical computer scientists and mathematicians, who know it can be so. From an engineer's point of view, changing arithmetic is catastrophic, but mathematics does not care how $1, 2, 3$ are represented, or if they are always the same – as long as there are no overlaps – and that is a description of simple one-to-many encryption so a mathematician intuits that encryption and arithmetic are connected. An engineer worries that an encrypted memory address changes during processing and misses its target, and it does, but that is 'hardware aliasing' [3], and how to compile for that is documented in software engineering literature [4] (reuse encrypted addresses, never recalculate, as that encrypts differently). The five fields that bear, computer engineering, security engineering, computer science, sotware engineering and mathematics, each cover a fraction of the subject area and we have to convince each expert of an unknown 4/5.

Having a working platform has helped us understand that all conventional machine code instruction sets are inherently insecure even in the encrypted context (see paragraph 2.10), although we initially set out to prove they could be made to work in it, and we describe here a machine code instruction set architecture that provably fixes the problem. Hopefully the message will get to manufacturers. It requires backing from a kind of randomised compilation, or it would always be subject to statistically-based 'dictionary attacks' based on the tendency of human programmers to prefer small numbers such as $0, 1, 2$, and that is described here too.

1.2 **Organisation.** The organisation of this article is as follows. Section 2 summarises processor design and working in this kind of encrypted computing, hoping to banish common (mis-) preconceptions. The reader should look there first. The basic processor design is as expected for the decades-old classic single pipeline RISC processor [5] and is not intrinsically of interest except for the modifications as described and that ordinariness is much of the point.

Contemporary efforts at encrypted computing are reported in Section 3. The processor model described in this paper is running 20,000 times as fast as the only really practical comparable option, and 150,000,000,000 times as

fast as IBM are achieving (of the order of 1 logical gate operation per second) in their experiments using homomorphic encryption [6], [7] as their approach to encrypted computing (homomorphic encryption gives rise to an example of an 'appropriately modified' arithmetic as referred to above).

Minor security engineering considerations arising from putting the principles into practice are described in Section 4. A reader should look there for hardware minutia. It should be noted by the way, that key management (a bugbear of security engineers) is not a concern of this paper, both by virtue of arguments given and because it can be done via extant hardware. It is the business of hardware engineers where to put keys in the processor and how to get them there, and the business of security engineers to ensure they do it right, but that has nothing to do with encrypted computing.

Performance is taken up in Section 5. Some account is given of what computer architecture approaches for improved performance work well and why. It has been found that a homomorphic encryption cannot work well in this context, for example (the encryptions that work well are fast symmetric encryptions), because classical pipelining[1] in the processor does not let one instruction start while another is running. The reason is that the 'execution' stage of an arithmetic instruction cannot be short-cut and it takes the full length of the pipeline so a depending following instruction must wait to start for the one ahead to finish, making the pipeline too long as well as just one-instruction-at-a-time.

Vulnerabilities of conventional instruction sets were explained in Section 2 and Section 6 shows a modified instruction architecture that makes encrypted computing provably secure. It requires, however, code to be compiled by an 'obfuscating' compiler that takes statistical advantage of its possibilities. The requirements for the compiler are sketched in that section (the working compiler's URL is given). We contend via this paper that processor, instruction set and compiler comprise a three-point system for provable security for the user against the operator in encrypted computing.

1.3 **What is new here?** This paper primarily puts forward an argument, but conventional benchmark results are reported for the first time. That has become possible only with the development of a significant software infrastructure (compiler, assembler, linker, debugging instruments, micro operating system) in support, representing about 25 person-years of software engineering effort. Measures carried over from the earlier report [8] have been bettered by 50% and we say what computer architecture optimisations have made that happen. Naysayers on the possibility of operating system support have been proved wrong, and the security concepts have meanwhile been proved correct at hardware-level [9] and in mathematical abstraction [2] and we are reporting that in summary here along with experimental evidence, to boost confidence in the practicality of the approach. A specially adapted instruction set architecture for encrypted working is shown here too and we report progress with the 'obfuscating compiler' for it. We did not think it would be possible, but almost all of C has been covered but for computed gotos, including C's extensive use of pointers.

1. Pipelining is the means by which procesors (and car factories) achieve speed-ups. Execution of an instruction (resp. building a car) is divided into stages carried out at different places in the processor (resp. car factory). One instruction takes just as long to execute – its *latency* – but as many as there are pipeline (resp. production line) stages may be worked on at the same time, resulting in a 10-20× speed-up overall (the *thruput*).

2. Reference Points

This section summarises the basic working of the processor architecture introduced in [8] and hopes to dispell misconceptions about processors that security engineers and software engineers seem to own, and misconceptions about abstract mathematics that through experience we have learned hardware engineers are prone to. We cannot anticipate everything, so we beg readers to consider that if there is something fundamental that they do not believe to consider that they are wrong in that, since they can compile the code, examine the matematical proofs, run the models, make up and run their own programs, etc, with the source code at the URLs given. Everything is open to inspection.

2.1 **Encryption is not done in software.** There are no timing or power drain measurement attacks. If encryption is needed, a dedicated hardware unit inside the processor pipeline does it, taking the same time every time, using no memory or anything else programmatically accessible.

2.2 **Memory is not part of a processor.** RAM is an external peripheral from a processor's point of view, like a keyboard or screen or disk, and the same considerations apply. Arguments should be based on that view.

2.3 **Architecture.** The prototype is based on the classic, simple, single pipelined RISC architecture of [5] taught in undergraduate computer architecture classes for decades. It is clocked at a nominal 1 GHz with 3 ns internal cache, separate for instructions and data, and separate for user and operator modes of working.

Processors contain registers for temporary storage. The layout and functionality follows the OpenRISC v1.1 architecture standard at http://openrisc.io ('OR1.1'). There are 32 general purpose registers (GPRs) and 65,536 special purpose registers (SPRs). Some SPRs' OR1.1-defined functions have had to be modified for security reasons and that is described in Section 4. Registers and buses are either 64 or 128 bits wide (it differs per processor model), holding either encrypted 32-bit data or unencrypted 64-bit data.

The pipeline incorporates basic optimisations such as speculative branch execution and prediction. Data is forwarded directly between instructions in different stages before it has first passed through the processor registers (saving one clock cycle). Data hazards (input to an instruction needs to wait for output from an instruction ahead) stall instructions. Successive models have incorporated cleverer features into the pipeline, such as on-the-fly instruction reordering, and their effectiveness is reported in Section 5.

2.4 **Modes.** There are two modes of working of the processor: *user* and *operator*, as per the OpenRISC specification. User mode works encrypted on data that is 32-bit beneath

the encryption and operator mode works on 32- or 64-bit unencrypted data. Supervisor mode has no access restrictions, while user mode is limited as described in Section 4.

2.5 **The ABI as attack vector.** The only means of attack considered here is via the machine code instructions understood by the processor, the syntax and semantics of which consutitute the 'application binary interface' (ABI). Physical attacks via scanning electron microscope are out, but rebooting, single-stepping user-mode, ('debugging') etc. are in. We the designers control the ABI, but it differs little from a standard OpenRISC ABI. There is no $x+y$ instruction, for example, but there is a $x+y+k$ instruction, where k is a constant embedded (encrypted) in the instruction.

2.6 **Adversaries.** The operator is the notional adversary who tries to read the user's data, and/or rewrite it. By 'the operator' is meant the operator mode of operation of the processor, in which instructions have access to every register and every memory location. Human agents such as operating system designers may have indirect and illicit access to operator mode via backdoors in the operating system or vulnerabilities in operator-mode applications such as servers on low IP port numbers or via malware introduced later, or access may be direct and outwardly 'legitimate', by the owner or administrator of the machine and/or operating system. Whatever the case, as the most privileged user on the machine, 'the operator' also stands in for all possibilities as a potential adversary. If we can secure user data from 'the operator', it is secure from all.

2.7 **A successful attack** consists of 'the operator' decrypting user data, or on the balance of probabilities deducing what it is likely to be, or modifying a user's program or interfering in it to produce data (when decrypted) that takes some intended value, or some range of intended values with heightened probability. The subtlety here is the 'intended'. It is not a successful attack to overwrite the user's data with zeros while the operator does not know yet what zero decrypts to. If they can say with probability above chance ($1/2^{32}$ for a single value) what the written value is when decrypted, then it is a successful attack. Producing the encrypted key, encrypted, even if the operator cannot yet read it, would be a successful attack (there exists a proof that the operator cannot do that particular thing by any deterministic or statistical method – the reader should refer to the final theorem in [2] for the general result that backs that deduction).

More subtlety is that it is not enough for a successful attack to work out (even only stochastically) what user data means. Instead the value of the (decrypted) data must be predicted. It might be guessed, for example, that the program is an image analysis algorithm because it passes through a certain point 1024×768 times, so the last time through probably represents a border pixel that is black. But 'black' does not tell what (decrypted) value represents blackness. The 'obfuscating compiler' here takes care to distribute the value across the full 32-bit range with equal probability.

2.8 **Simulation.** Our results are verified by simulation. The open source OpenRISC 'Or1ksim' simulator, available from opencores.org/or1k/Or1ksim, was the original source of the simulation source code, so the implementation of OpenRISC instruction semantics departed from an initially correct base.

That has been modified to a cycle-accurate pipeline simulation over three years and eight processor prototypes. Pipeline measurement accuracy is inherent in such simulation because the original instruction semantics were split into multiple decode, prep, read, execute, post and write steps that take one hardware clock cycle each, and the pipeline moves all instructions on one stage per cycle at the same time, insofar as they may be moved – sometimes an instruction cannot proceed for lack of needed data (from memory or from an instruction ahead) or a stalled instruction ahead, and in that case they stall the instructions behind too. We count how many cycles a program takes in our benchmarks. Roughly 50% of pipeline slots are unoccupied on average (see Section 5) and it is not unusual for a cycle to run without any instruction terminating (i.e., exiting the pipeline) that cycle. The point is that it is not possible to 'measure wrong' this way. The open source code archive including a full log history is at http://sf.net/p/or1ksim64kpu.

2.9 **Instruction set.** In user mode, the processor models here natively run the 32-bit OpenRISC instruction set modified for encrypted operation. The modification is chiefly that any 'immediate' data embedded in an OpenRISC instruction must be in encrypted form. Because encrypted data is too long to fit in the standard's 16-bit immediate data field, an extra prefix instruction has been introduced in order to allow encrypted immediate data to be started in a prefix to the instruction, and in a prefix to that, if necessary.

Other instruction set modifications are minor. The displacement (as in an array) from the base address embedded in a load or store instruction must be zero in user mode, in order to reduce the arithmetic required in-pipeline. An extra displacement register is ignored in a 'move to/from SPR' instruction in user mode because it would give away the encryption directly. A shift instruction has its fields reordered so the shift is contiguous. That is all.

The 32-bit floating point as well as integer instructions have been fully implemented for user mode. That makes the platform logically 32-bit, even though physically it is 128-bit (or longer, in principle). The compiler implements 64-bit (or longer) data and operations for user-mode with the 32-bit instructions, following standard compile practice.

In operator (unencrypted) mode, the native OpenRISC instructions for 64-bit operations are available as well.

2.10 **Security of encrypted computation.** Adapting all the standard OpenRISC instruction set for encrypted working has answered questions about what programs can run in this context, and particularly that it is not hard to write (unencrypted, operator mode) operating system support for user programs running encrypted. The operating system does not need to know what the decrypted value of a user mode datum is (e.g., in memory fault handlers and I/O routines). But the experience has clarified that standard instruction sets are inherently insecure [10] with respect to the operator as adversary, who may steal an (encrypt-

ed) user datum x and put it through the machine's division instruction to get x/x, which is an encrypted 1. It is then simple to construct any encrypted y by applying the machine's addition instruction to get (encrypted) $1+1$, etc. By comparing the encrypted $1, 2, 4,$ etc. obtained with any encrypted z using the instruction set's comparator instructions (testing $2^{31} \leq z$, $2^{30} \leq z$, ... in turn and subtracting whenever a test succeeds), the decrypted value of z may be deduced. Part of the contribution described in this paper has been to develop a 'FxA' instruction set for encrypted RISC for which it may be proved that there is no method of decryption like that above, or any method that is statistically right at a level above chance (see Section 6).

2.11 **Encryption.** The processor models have been tested while embedding Rijndael-64 and -128 symmetric ciphers (the latter is the US Advanced Encryption Standard (AES) [11]), RC2-64 [12] and Paillier-72 [13]. The last is an additively homomorphic[2] cipher that runs without keys in the processor. In principle, any block cipher with block size that fits in a machine word could be integrated. The encryption hardware is in the pipeline, taking up the majority of stages. For the symmetric encryptions the hardware consists of an encryption/decryption unit. An AES encryption/decryption round takes 1 ns and fits in one hardware stage.[3] The complete AES encryption/decryption unit occupies 10 stages. For other symmetric ciphers, between 1 and 20 pipeline stages are configured in the simulator. For homomorphic encryptions, multistage arithmetic occupies that space.

Note that all the encryptions, whether symmetric or homomorphic, are one-to-many. For symmetric encryptions, pseudo-random padding under the encryption is generated by hashing together the inputs' paddings and the instruction, allowing a trace to be audited. For Paillier, 'blinding' multipliers are generated instead.[5]

The choice of encryptions has been dictated by the development path. The Or1ksim simulator was first expanded from the original's 32 bits to 64 (plus the changes that made it cycle-accurate and pipelined) and at that point 64 bit ciphers could be handled. An extra two 32-bit prefixes per instruction are required to hold 64 bits of encrypted immediate data. That prefix configuration is also sufficient to hold 72 bits of encrypted data, so Paillier-72 could be accommodated without further alteration to the toolchain, but required a doubling of path widths from 64 to 128 bits in the processor models. AES-128 then became possible, requiring four 32-bit prefixes per instruction and the toolchain to be modified again.

Paillier-72 is insecure for practical purposes but it has served to investigate the use of a homomorphic encryption in this setting. Paillier does not become as secure as AES-128 until about 2048-bit blocks are used, but 2048-bit Paillier arithmetic needs infeasibly many stages for the processor pipeline. Nevertheless, the closest extant design is HEROIC [15], [16] (see Section 3, a stack machine running encrypted with a 'one instruction' machine code (the 'OI' in HEROIC) with 2048-bit words encrypting 16 bits of data each. It does the 2048-bit Paillier arithmetic in hardware, so it is possible (HEROIC simply takes 4,000 cycles of the base hardware for each arithmetic operation).

2.12 **Toolchain.** The existing GNU 'gcc' v4.9.1 compiler at http://github.com/openrisc/or1k-gcc and 'gas' version 2.24.51 assembler at http://github.com/openrisc/or1k-src/gas ports for the OpenRISC 1.1 architecture have been adapted for the encrypted instruction set. The source code for the modified compiler is at http://sf.net/p/or1k64kpu-gcc and that for the modified assembler is at http://sf.net/p/or1k64kpu-binutils. Only the assembler, not the compiler, needs to know the encryption key. Executables are written as standard ELF format files. An experimental 'obfuscating' compiler, as described in Section 6, is also coming on-line. Global variables are a point of difficulty for both compilers and, currently, explicit link-table entries for them have to be provided. The difficulty is that the encrypted address needs to be shared between different compilation units, but encryption takes place in the assembler, after compilation. Our stop-gap solution is to impose a common address via link-table entries, or port the code to remove globals. The difficulties of providing toolchains for encrypted computation should not be underestimated, but they are soluble.

2.13 **Limits.** Word width/encryption block size up to 2048 bits is contemplated for current technology. Memory paths would have to be appropriately broadened from the current 64 bits, and memory accesses paralleled correspondingly.

2.14 **Testing configurations.** Tests are centered about a 15-stage pipeline configuration with a nominal 1 GHz clock. Of that, 10 stages are for the modified arithmetic/encrypton hardware, but between 1 and 20 stages have been mapped. Memory and cache latencies (typically 13.5 ns, 3 ns respectively) are adjustable for testing. Although the data memory path is always a full word (64 or 128 bits or more) wide, the program memory path is restricted to 64 bits, which results in two 32-bit instructions per cycle being pulled to a split 32-word look-ahead/behind buffer in the pipeline fetch stage. The relatively narrow (64-bit) instruction path is a continuing bottleneck that we cannot easily do away with because of legacy issues.

2.15 **Key management.** There is no means to read keys once they have been embedded in the processor, where they configure the hardware functions. Keys must either be embedded at manufacture, as with Smart Card technologies [17] or introduced via a Diffie-Hellman circuit [18] or equiv-

2. 'Homomorphic' for Paillier means multiplication of encrypted numbers corresponds to addition of the unencrypted numbers.

3. The Intel/AMD 'ASENC' AES round instruction has latency 0.95ns (4 cycles at 4.2GHz) on Skylake cores (Table C-9 of [14]), so one round in one pipeline stage at 1 GHz is very achievable.

alent that loads the key in public view without revealing it.[4]

Hopefully, the basics are now covered for the reader.

3. Related platforms

In this section, the state of the art is discussed and compared.

3.1 **HEROIC** [15] is the most comparable contemporary. It is a 16-bit processor working in Paillier-2048 encryption [13] on a stack-based architecture. Its core does encrypted addition in 4000 cycles and 20 μs on 200 MHz programmable hardware. That is equivalent to a 25 KHz Pentium.

Stack machine architectures differ from conventional von Neumann architectures and are not currently manufactured, but there have been hardware prototypes in connection with Java [19], [20] in the past. Like the processor discussed in this paper, HEROIC works by substituting a modified arithmetic. Its basis is the replacement of conventional 16-bit addition by multiplication of 2048-bit encrypted numbers modulo a 2048-bit modulus m, and other operations are done in software using addition as the primitive. The Paillier encryption $\mathscr{E}$ is used. It has the 'homomorphic' property that multiplying encrypted numbers $\mathscr{E}(x)$, $\mathscr{E}(y)$ is the same as adding the unencrypted numbers x, y:

$$\mathscr{E}(x) * \mathscr{E}(y) \mod m = \mathscr{E}(x+y) \qquad (+)$$

A difficulty with (+) for encrypted computing is that the addition on the right is not the addition mod 2^{16} of 16-bit computer arithmetic. The result has to be 'renormalised' to a remainder mod 2^{16} under the encryption, which accounts for half the cycles taken by HEROIC for the encrypted addition. The microcode subtracts 2^{16} and looks up a 'table of signs' to see whether the encrypted result is negative or positive. In order to facilitate that, HEROIC encryption is one-to-one, not one-to-many, or the lookup table would be too large.[5] At $2^{16} \times 2048$ bits, it is already 16 M bytes in size The 'selling point' of Paillier is that (+) means that the modified arithmetic in the processor *needs no keys*. But, despite the headline, the table of signs amounts to a secret key per user.

That implementation is also used in our Paillier-based processor models, except that, at $2^{32} \times 72$ bits times the number of aliases in our one-to-many encryption, the table of signs is too large to site locally with current technology, so signs are calculated remotely on demand and cached locally, and the time taken in the remote calculation is not counted.

3.2 **Ascend** [21] obscures instructions and data from the operator's view by a variety of means, both cryptographic and physical. The processor protects code on the way to the processor via encryption. I/O is encrypted and the processor runs in 'Fort-Knox'-like isolation, matching pre-defined statistics on observables. Communication with memory is encrypted too, via 'oblivious RAM' [22], [23], [24].

The idea of physical isolation plus encrypted memory has emerged many times over the years (e.g., [25], [26]) and success means doing it as well as Ascend does. Otherwise channels such as cache-hit statistics [27] and power drain [28] can give away information to a privileged observer. Ascend runs RISC MIPS instructions [29] and slows down by a factor of 12-13.5$\times$ in encrypted mode with AES-128 (absolute speeds are not given in [21]), as compared to 10-50% slowdown for our models (Section 5).

3.3 **Intel's SGX**$^{\text{TM}}$ ('Software Guard eXtensions') processor technology [30] is often cited in relation to secure or encrypted computation in the cloud, because it enforces separations between users. However, the mechanism is key management to restrict users to memory 'enclaves'. While the enclaves may be encrypted because there are encryption/decryption units on the memory path, that is encrypted and partitioned storage, a venerable idea [25], [26], not encrypted computing. RAM is a peripheral to a processor.

SGX machines are used [31] by cloud service providers where the assurance of safety is a selling point. But the assurance is founded in the customer's trust in electronics designers 'getting it right' rather than mathematical analysis and proof, as for our and HEROIC's technologies (see Section 6). There are subtle ways for engineering to leak secrets, e.g. via timing variations and power use [27], and there have recently been successful attacks against SGX based on timing [32]. Moreover, use of the enclave areas is optional for the software author, and it is up to the author which sections of code to run in enclave and which not to, meaning that no guarantees can be made on the basis of the hardware. The security of every piece of software depends on the software.

3.4 **IBM**'s efforts at making practical encrypted computation using very long integer lattice-based fully homomorphic encryptions (FHEs; additively *and also* multiplicatively homomorphic) based on Gentry's 2009 cipher [7] deserve mention. An FHE $\mathscr{E}$ extends the Paillier equation (+) to cover multiplication on the right too. However, it is single bit arithmetic, not 16- or 32-bit arithmetic under the encryption. The single bit operations currently take of the order of a second each [33] on customised vector mainframes with a million-bit word size, about the speed of a 0.03 Hz Pentium, but it may be that newer fully homomorphic ciphers based on matrix addition and multiplication [34] will be more practical. The product will never be capable of arbitrary general purpose computation in any case, just certain finite calculations. The obstacle to computational completeness is

4. There is no direct consequence of running with the wrong key in the machine: if user A runs with user B's key in the machine, user A's program will produce rubbish, as the processor arithmetic will be meaningless with respect to it; if user A runs user B's program with user B's key in the machine, then the output will be encrypted for user B's key, and the input will need to be encrypted in user B's key, which user A can neither supply nor understand. Security depends not on access to encrypted numbers but on other factors, such as whether A, who may be the operator, can leverage observed computations that use B's key to learn about the encryption, and that question is answered in Section 6 – negatively, for the right instruction set.

5. Paillier may embed random 'blinding factors' into encrypted numbers. Those are multipliers r^n mod m, where $n{=}pq$ and $m{=}n^2$ is the public modulus. Paillier decryption involves raising to the power of the order $\phi{=}(p{-}1)(q{-}1)$ of the multiplicative group mod n, so r^n becomes $r^{\phi n} = (1{+}kn)^n{=}1{+}kn^2{+}\ldots{=}1$ mod n^2 and does not affect the decrypted value. HEROIC's one-to-one encryption does not use different blinding factors.

the same as that which HEROIC overcomes: an encrypted comparison operation is needed for practical operation with potentially unbounded computations, but HEROIC's 'table of signs' solution is not feasible for a million-bit encryption.

Sometimes applications require a fixed small number of multiplications and then a *somewhat homomorphic encryption* (SHE) may do instead of a FHE. A SHE is without the periodic renormalisations that are the hallmark of Gentry-style FHEs, resulting in faster and smaller encryptions. Without renormalisation, arithmetic eventually takes numbers out of range, for example nearing $2m$ in a calculation mod m. But before that happens the calculation will have finished. Lauter et al. [35] quote a scheme with block-size about 43.5KB and encrypted 1-bit addition in 1ms, multiplication in 43 ms, achieved on a 2.1 GHz Intel Core 2 Duo in 1 GB of RAM. That equates to a 30Hz (sic) Pentium.

3.5 **Moat electronics.** Classically, information may leak indirectly via processing time and power consumption, and 'moat technology' [36] to mask those channels has been developed for conventional processors. The protections may be applied here too, but there is really nothing to protect in terms of encryption as encrypted arithmetic is done in hardware, always taking the same time and power. There are separate user- and operator-mode caches in our models, and statistics are not available to the other mode, so side-channel attacks based on cache-hits [37], [27] are not available.

3.6 **Oblivious RAM** [22], [23], [24] and its evolutions [38], [39]) is often cited as a defense against dynamic memory snooping. That is in contrast to static snooping, so-called 'cold boot' attacks [40], [41], [42] – essentially, physically freezing the memory to retain the memory contents when power is removed, against which HEROIC, SGX and our technology automatically defend because memory content is encrypted; addresses are also randomised in our case. An oblivious RAM remaps the logical to physical address relation dynamically, taking care of aliasing, so access patterns are statistically impossible to spot. It also masks the programmed accesses in a sea of independently generated random accesses. However, it is no defense against an attacker with a debugger, who does not care where the data is stored. It provides no defense against the operator and operating system, which the technology here can be proved to do (Section 6).

Some 'oblivious' behaviour is naturally present in the processor described in this paper, because data addresses are (nondeterministically) encrypted and so the address encryptions naturally vary dynamically at runtime.

4. Engineering for security

This section deals with special adjustments in the hardware required with respect to the basic OpenRISC architecture and specification in order to accommodate the 'encrypted computing' mode of working and security.

First of all, from an application programmer's point of view, it should be emphasised that in (encrypted) user mode, the processor executes the 32-bit instruction subset

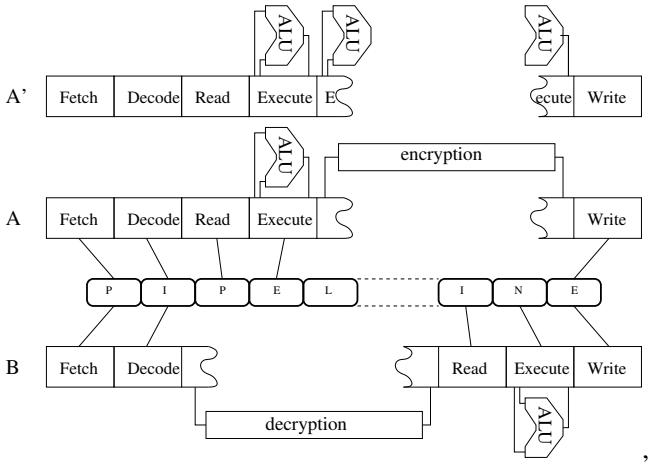

Figure 1. The pipeline is configured in two different ways, 'A' and 'B', for two different kinds of user mode instructions during encrypted working, in order to reduce encryption/decryption to at most once per instruction (this illustration of the principle is adapted from [8]).

of OpenRISC as normal and that is all that the programmer needs to trust in. The compiler will take care of it, modulo some issues of the current state of our toolchain that can make porting source codes nontrivial (e.g., global variables are problematic when used in different compilation modules, requiring explicit link table inputs to tie the compilation together).

For a library or systems programmer, the situation is nearly the same. User mode coverage of 32-bit integer and floating point OpenRISC instructions is complete, and OpenRISC's application binary interface is supported. Access to most SPRs in user mode is already denied by OpenRISC – e.g., those that configure the memory map – but there are some further SPRs put off limits to protect the encryption.**??**

4.1 **Access denials** for user mode beyond OpenRISC standards are to the processor version number SPR, the floating point unit control SPR and the performance statistics SPRs. Certain bits in the status SPR (such as the one that is stuck at 1) are also denied. The timer tick SPR is aliased so there is a different timer tick per processor mode and it is not the same between them, so plaintext and encrypted readings of the same clock cannot be obtained. User mode access is also denied to the cache control SPR (prefetch, flush), but in any case, cache is aliased so there is a different cache per processor mode and no back-channel communication is possible by that route and we are experimenting to determine if user mode cache control may be allowed after all.

4.2 **Supervisor mode** (recall: unencrypted), which the operator has access to, allows all instructions and unrestricted access to registers and memory. Supervisor mode instructions can read user data in memory, but it is in encrypted form. Likewise, user data in registers appears in encrypted form to operator mode via a hardware protocol (*) sketched below.

The operator mode instruction pipeline, which in principle is the classical 5-stage fetch, decode, read, execute, write

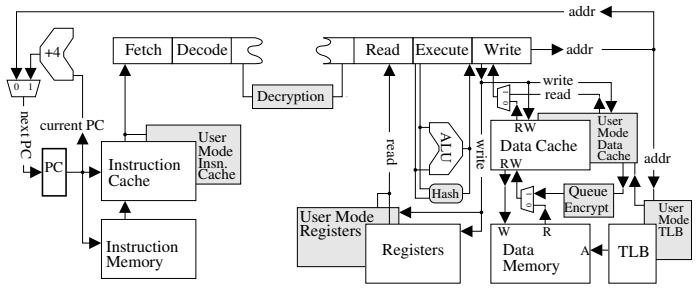

Figure 2. Pipeline integration, showing shadow units for user mode. This illustration adapts drawings in [8].

pipeline of a RISC processor, lies embedded in a longer pipeline containing the encrypted arithmetic stages for user mode. There are two major 'tricks' of implementation for good performance with symmetric encryptions, both based on the necessity to decrypt, do the arithmetic, then encrypt again, in order to carry out the 'encrypted arithmetic' that underpins encrypted computing with such encryptions. The first trick is that *the pipeline is configured in two ways*, 'A' and 'B', for encrypted running. That is illustrated in Fig. 1. The pipeline stages are named in exploded view respectively above (A) and below (B) the pipeline in the figure. The encryption/decryption stages ('codec') are shown in abbreviation, but they consist of one stage per round of the encryption (10 for AES). A conventional ALU figures in only one stage in both configurations, and its positioning in the pipeline relative to the codec is the focus of the figure.

4.3 **The dual configuration** pipeline is because, for symmetric encryptions, there is only room for one multi-stage encryption/decryption unit in the hardware and some instructions need it before the arithmetic stage(s), and some need it after. The memory load and store instructions would conventionally need it twice, once for the address displacement sum and once for the data transfer to/from RAM, but they are restricted in user mode to a zero 'address displacement' field (that slightly impacts program efficiency, but not functionality), which means they need the unit just once per instruction, for the data transfer. User data is kept encrypted in RAM by the protocol (*) described in 4.4 that manages the transition between user and operator mode in the processor. The 'A' configuration is for instructions that use the encryption/decryption unit after arithmetic, the 'B' configuration when it is the other way around.

There are data hazards generated by conflicts between the two pipeline configurations for the single codec and other single functional units, but those are not very significant, as the measurements of Section 5 show, and they can be obviated entirely by physically doubling the number of units. Section 5 enables rational decisions on doing that.

4.4 **The arithmetic logic unit** (ALU) comprises the second trick of implementation for good performance. In order to reduce the frequency with which the encryption/decryption unit is used in association with symmetric encryptions, ALU operation is *extended in the time dimension*, so it covers a series of consecutive (encrypted) arithmetic operations in user mode. The first of the series is associated with a decryption event and the last with an encryption event (both the two 'tricks' are described in [8]; note that by 'arithmetic operations' is meant the arithmetic stages of individual instructions, not the whole instructions). In support:

A different set of registers is aliased in for each mode (*)

The user mode registers 'shadow' the operator mode registers, as illustrated in Fig. 2, where the shadow units are shown slightly behind the non-shadow (i.e., operator mode) units. Where the operator mode register contains the encrypted value or a placeholder, the user mode register contains the unencrypted number. In user mode, arithmetic is carried out in shadow registers, which contain decrypted values. The protocol (*) is mathematically proved to maintain invariants that secure operator mode from accessing data unencrypted that originates in user mode, and vice versa.[??] The protocol also assures user data is stored encrypted in memory, never exposed in unencrypted form.

4.5 **Key management** is not an issue in itself, by the following argument. Firstly, changing the encryption key on a change of user empties the shadow registers so that one user does not have access to another's unencrypted data in registers. Memory always contains encrypted user data that cannot be read by the newcomer because it is encrypted with a different key. Secondly, if something does go badly wrong and the encryption key is not changed on change of user, then the shadow registers containing the unencrypted data are preserved and data in them is potentially vulnerable. However, the argument in the footnote[4] says B has no direct access to A's unencrypted data or program even in this situation. I/O is still encrypted with A's key.

The danger is that B might deduce indirectly what the data in registers is by running a branching comparator instruction. B's problem is what to compare with, as B has no knowledge of constants in A's encryption. Unfortunately, yes, as already argued, with the OpenRISC instruction set B, can create an encrypted 1 via x/x from nearly any encrypted x of A's, and use it to build constants to order. But that attack works in any context, not only this particular one, and the fault is the instruction set's. Using an 'FxA' instruction set (Section 6) cures the problem.

4.6 **Cache** raises similar questions and is treated similarly to registers. As remarked, separate data and instruction caches are aliased in per processor mode, just as for registers. On change of user, the user cache is flushed. As already argued for register content, a change of user from A to B without flushing the caches is not in itself insecure, even if it mistakenly happens. Any I/O is still encrypted in A's key. The danger is tests that B may be able to run on A's cached data if B gets control, and the defence against that depends on the instruction set, as discussed in Section 6.

4.7 **Further modifications** to conventional processor design include an address translation look-aside buffer (TLB) in two parts. A conventional 'back-end' to the TLB fills the

conventional role of remapping addresses page-wise, but the 'front end' is organised by single word addresses, not pages, and its job is to remap encrypted addresses to the physically backed range in first-come, first-served order. Since data that will be accessed together tends to be accessed together for the first time too, this enables cache readahead to continue to be effective even though encrypted addresses are spread randomly over the whole cipherspace. The TLB front-end is eventually a limiting overhead, but it does not affect programs at all when their footprint fits in cache, which is sensible software design.

5. Performance

The original Or1ksim OpenRISC test suite codes (written mostly in assembler) were used as benchmarks for encrypted running in [8]. Most modern performance benchmark suites are unavailable because they rely on external library support such as linear programming packages and math floating point libraries, as well as faithful system library routines such as 'printf', all of which must be written and debugged. If those could be ported to compilable code in good time, debugging would take months (the ported *gcc* compiler inherits known bugs, such as sometimes not doing switch statements right, sometimes not initialising arrays right, etc.). In particular, the well-known 'spec' benchmark suite is unavailable because its source code is commercially protected. Some standard but less evolved, more standalone, benchmarks have been got running, such as Dhrystone 2.1.

Table 1 details a base level performance for our prototype in the instruction set add test of the Or1ksim suite, with RC2 64-bit symmetric encryption and before hardware improvements for performance have been applied. The 64:16:20 mix for arithmetic:load/store:control instructions (no-ops and prefixes discarded) compares to the 60:28:12 mix in the standard textbook [43]. This baseline is 4% better than the numbers reported in [8]. The test spends 52.7% of its time in user mode, as against 54.8% in [8] (4% is 2.1/54.8). Pipeline occupation is $1-20.7/52.7=60.7\%$ in encrypted mode, for 607 Kips (instructions per second) at 1 GHz clock.

The same test with Paillier-72 (128-bit architecture) gives worse performance, as some of the arithmetic is done in software. Table 2 compares RC2 with Paillier on the 'add test'.

Table 2

add test	cycles	instructions
RC2 (64-bit)	296368	222006
Paillier-72	438896	226185

The difference is principally due to more pipeline stalls, because Paillier arithmetic always takes the length of the pipeline to complete in. There is no such thing as internal feed forward for the addition carry bit, for instance, because the result is entirely unknown until it is complete. That stalls following instructions that need the result almost until the instruction ahead has exited, leaving most of the pipeline empty. The disparity increases on swapping addition for multiplication, which is entirely done in software. Table 3 compares RC2 with Paillier on the 'multiplication test':

TABLE 1. BASELINE RC2 (64-BIT SYMMETRIC ENCRYPTION) PERFORMANCE, OR1KSIM 'ADD TEST': PROPORTION FINISHING PER CYCLE.

RC2: cycles 296368, instructions 222006		per cycle	
	mode	user	super
arithmetic	register instructions	0.2%	0.2%
	immediate instructions	7.8%	9.8%
memory	load instructions	1.0%	3.0%
	store instructions	1.0%	0.0%
control	branch instructions	1.1%	5.2%
	jump instructions	1.2%	5.1%
	sys/trap instructions	0.5%	0.0%
	no-op instructions	7.3%	16.8%
	prefix instructions	11.8%	0.0%
	move from/to SPR instructions	0.1%	2.8%
	wait states	20.7%	4.4%
	(stalls)	(17.4%)	(3.7%)
	(refills)	(3.3%)	(0.7%)
	total	52.7%	47.3%

Branch Prediction Buffer			
hits	10328 (55%)	misses	8219 (44%)
right	8335 (44%)	right	6495 (35%)
wrong	1993 (10%)	wrong	1724 (9%)

User Data Cache			
read hits	2942 (99%)	misses	0 (0%)
write hits	2933 (99%)	misses	9 (0%)

Table 3

mul. test	cycles	instructions
RC2 (64-bit)	235037	141854
Paillier-72	457825	193887

In contrast, performance with symmetric encryptions is very sensitive to forwarding along the pipeline, allowing instructions to close up to each other without waiting for an intermediate write to and read from registers. Table 4 shows that 33% of processor speed is due to forwarding, while on-the-fly instruction reordering gives another 3%:

Table 4

add test		forwarding	
RC2 (64-bit) cycles		✓	×
reordering	✓	296368	412062
	×	315640	441550

Paillier has little sensitivity to forwarding. That is expected because an arithmetic result is not available before the penultimate stage. The only prospect for improving Paillier speed appears to be to compile multithread programs, so there may be instructions behind that can overtake a stalled instruction.

We have devised three performance optimisations tailored to the architecture and the bottlenecks noted: (a) instructions with trivial functionality in the execute phase

(e.g., 'cmov,' the 'conditional move' of one register's data to another) but stalled in read stage have been allowed to speculatively proceed on the assumption that they will be able to pick up the data via forwarding later during their progress through the pipe[6]; (b) the fetch stage has been doubled to get two instructions per cycle and catenate the prefix instruction to the instruction they prefix instead of taking up pipeline slots in their own right; (c) a second pipeline has been introduced to speculatively execute both sides of a branch at once.

5.1 **Flexible staging** (a) takes the cycle count in the 'add test' down from 296368 to 259349 cycles. It is very effective, as may be expected. Innovations (b) and (c) then contribute as shown in Table 5:

Table 5

add test RC2 (64-bit) cycles		deprefixing (b) ✓	deprefixing (b) ×
branch both (c)	✓	237463	257425
	×	241992	259349

5.2 **Deprefixing** (b) is intended to beat the bottleneck caused by the fact that, in user mode, much of the instruction code real estate is taken up by embedded (encrypted) constants, which do not have any functional activity. They take time to load to the processor and without deprefixing what is notionally a single instruction would be spread 32 bits at a time across several stages of the pipeline.

Deprefixing instead binds the embedded constant to the instruction metadata as the sequence of prefixes and opcode enters the pipeline, reducing each instruction to a unit occupying a single pipeline stage. To keep the pipeline filled, the fetch rate has been increased and a 16×32-bit instruction buffer is scanned by the decode stage. The last 16 32-bit instructions read are also retained in case of a tight loop.

The mechanism can sometimes slow down the pipeline, because instruction opcodes that are on average closer together in the pipeline make data hazards more likely. A mispredicted branch also aborts more partially executed instruction units than it would have otherwise.

5.3 **Branching both ways** (c) is not effective in this test because only 3717 branches were predicted wrongly, but harder to predict branching code should benefit strongly.

Tables 2-5, though compiled with RC-64, also provide baselines for AES-128 via the Dhrystone 2.1 benchmarks shown in the first group of three lines in Table 6:

6. The 'assumption' is logically impeccable: the data needed must be supplied by an instruction ahead, which will finish before this instruction does and therefore furnish the data while it is still moving through the pipeline.

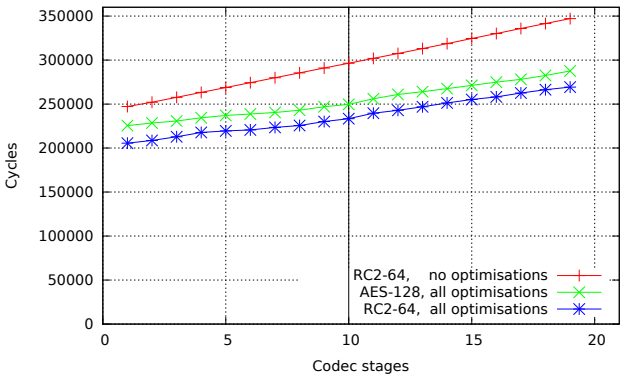

Figure 3. Number of executed cycles with symmetric encryption for the 'add test' of Table 1 against number of stages occupied by the multistage encryption/decryption unit ('codec'), showing 2.5% extra cost per stage.

Table 6

Dhrystone v2.1	RC2 (64-bit)	AES (128-bit)	None (32-bit)
Dhrystones per second	246913	183486	350877
VAX MIPS rating	140	104	199

Dhrystone v2.1 (gcc 4.9.2)	Pentium M 32-bit 1GHz O0	O2	O6
Dhrystones per second	735294	1470588	2777777
VAX MIPS rating	418	836	1580

Dhrystone v2.1	ARM250	ARM926	ARM1176
MHz	12	200	772
Dhrystones per second	12300	386540	1695505
VAX MIPS rating/GHz	583	220	965

That is to say, the numbers in Tables 2-5 should be divided by $104/140 \sim 0.75$ to get numbers for AES-128. The slowdown for 128-bit AES over 64-bit RC2 is mostly due to the 4, not 2, prefixes for an immediate constant in an instruction carrying immediate data. It illustrates that compilers for encrypted instruction sets should prefer to avoid inline data. For comparison, the benchmark run in operator mode, without encryption, is shown in the final column of the first three lines of the table. Immediate instructions comprise about half of all instructions in the benchmark mix, and attaching two or four prefixes to each has a marked effect, even though the 'deprefixing' optimisation is in the hardware. It is not known yet why it does not do better.

By the Dhrystone measures, the AES-128 prototype runs as a 330 MHz classic Pentium, or 250 MHz Pentium M. The list at http://www.roylongbottom.org.uk/dhrystone%20results.htm, shows a Pentium M is rated at 523 MIPS/GHz and a classic Pentium is rated at 322 MIPS/GHz. Thus, the RC2 prototype equates to a 433 MHz classic Pentium or a 266 MHz Pentium M. However, the results are compiler-sensitive, as seen by the variation through optimisation levels 0-6 in the middle rows of Table 6, and our compiler is rudimentary.

The final group of four lines in Table 6 shows the manufacturer's own MIPS ratings for different ARM chips, ranging from the ARM250 (pre-1990) to the ARM1176 (year 2003). ARM machines run a RISC instruction set so the benchmarks are more directly comparable with our

TABLE 7. AN FXA MACHINE CODE INSTRUCTION SET FOR WORKING WITH ENCRYPTED DATA

fields	*semantics*
add $r_0\ r_1\ r_2\ [k]_{\mathscr{E}}$	add $r_0 \leftarrow [[r_1]_{\mathscr{D}} + [r_2]_{\mathscr{D}} + k]_{\mathscr{E}}$
sub $r_0\ r_1\ r_2\ [k]_{\mathscr{E}}$	subtract $r_0 \leftarrow [[r_1]_{\mathscr{D}} - [r_2]_{\mathscr{D}} + k]_{\mathscr{E}}$
mul $r_0\ r_1\ r_2\ [k_0]_{\mathscr{E}}\ [k_1]_{\mathscr{E}}\ [k_2]_{\mathscr{E}}$	multiply $r_0 \leftarrow [([r_1]_{\mathscr{D}} - k_1) * ([r_2]_{\mathscr{D}} - k_2) + k_0]_{\mathscr{E}}$
div $r_0\ r_1\ r_2\ [k_0]_{\mathscr{E}}\ [k_1]_{\mathscr{E}}\ [k_2]_{\mathscr{E}}$	divide $r_0 \leftarrow [([r_1]_{\mathscr{D}} - k_1) / ([r_2]_{\mathscr{D}} - k_2) + k_0]_{\mathscr{E}}$
xor $r_0\ r_1\ r_2\ [k_0]_{\mathscr{E}}\ [k_1]_{\mathscr{E}}\ [k_2]_{\mathscr{E}}$	excl. or $r_0 \leftarrow [([r_1]_{\mathscr{D}} - k_1) \hat{} ([r_2]_{\mathscr{D}} - k_2) + k_0]_{\mathscr{E}}$
...	
mov $r_0\ r_1$	move $r_0 \leftarrow r_1$
beq $r_1\ r_2$ j $[k]_{\mathscr{E}}$	skip j instructions if $[r_1]_{\mathscr{D}} = [r_2]_{\mathscr{D}} + k$
bne $r_1\ r_2$ j $[k]_{\mathscr{E}}$	skip j instructions if $[r_1]_{\mathscr{D}} \neq [r_2]_{\mathscr{D}} + k$
blt $r_1\ r_2$ j $[k]_{\mathscr{E}}$	skip j instructions if $[r_1]_{\mathscr{D}} < [r_2]_{\mathscr{D}} + k$
bgt $r_1\ r_2$ j $[k]_{\mathscr{E}}$	skip j instructions if $[r_1]_{\mathscr{D}} > [r_2]_{\mathscr{D}} + k$
ble $r_1\ r_2$ j $[k]_{\mathscr{E}}$	skip j instructions if $[r_1]_{\mathscr{D}} \leq [r_2]_{\mathscr{D}} + k$
bge $r_1\ r_2$ j $[k]_{\mathscr{E}}$	skip j instructions if $[r_1]_{\mathscr{D}} \geq [r_2]_{\mathscr{D}} + k$
b j	skip j instructions unconditionally
...	

Legend: the r are register indexes or memory locations, the k are 32-bit integers, the j are instruction address increments, '$\leftarrow$' is assignment. The function $[\cdot]_{\mathscr{E}}$ represents encryption, $[\cdot]_{\mathscr{D}}$ decryption.

prototype's than are Intel's. The ARM926 (year 2000) model is about as fast as our prototype running unencrypted. Those numbers indicate that our simulation is producing sensible results. However, compiler details are not available, and the table has already shown that can make $\times 2$ or more difference, so ARM's numbers should be regarded cautiously.

In any case, performance results may be extrapolated as needed: Fig. 3 shows each extra pipeline stage costs 2.5%, both with AES and RC2 encryptions.

6. FxA Instruction Set

Standard instruction sets are insecure for encrypted working (recall the argument in 2.10 that x/x gives an encrypted 1, which then can be used to generate any desired encrypted value via addition), but the minimal 'one instruction' HEROIC instruction set turns out to be immune to the problem.

Denote by a *fused anything and add* (FxA) instruction set architecture one in which the compiler is able to displace the operands x_1, x_2 arbitrarily via constants k_1, k_2 embedded in the instruction, and also displace the result by a constant k_3. So FxA multiplication does:

$$(x_1 - k_1) * (x_2 - k_2) + k_3$$

That principle governs the design of all instructions. Most of a complete FxA instruction set for encrypted working is shown in Table 7. Each arithmetic instruction needs at least one embedded (encrypted) constant, by which the compiler may exercise control. Some FxA instructions, e.g. addition, need only *one* constant, as

$$(x_1 - k_1) + (x_2 - k_2) + k_3 = x_1 + x_2 + (k_3 - k_1 - k_2)$$

HEROIC's instructions are a (tiny) subset.

FxA instructions are secure with respect to several cryptographic notions. First of all, suppose that the processor enforces the rule that *no collisions* are possible between (i) encrypted constants that appear in instructions and (ii) runtime encrypted data values in registers or memory. For symmetric encryptions, our prototype can do that via different pseudo-random padding under the encryption. For a homomorphic encryption like Paillier, a different set of 'blinding' factors (a multiplier of the encrypted value that vanishes without effect during the decryption process) may be used. Then:

Fact 1. *There is no deterministic method by which the operator can read a program C built from FxA instructions, nor alter it to give an intended encrypted output.*

The supporting argument[??] depends on the operator, who is the adversary in this scenario, not being able to assign any particular meaning to observed changes in encrypted runtime data or instruction constants. That is true for an encryption with a relatively long expected time between re-occurrences of the same encrypted values for some same underlying unencrypted values (such as AES-128). However, HEROIC's one-to-one encryption maps collisions to equalities underneath the encryption, invalidating the assumption.

Moreover, the probability of guessing correctly that an encrypted constant in an instruction is a 1 or 0 is high whenever a human being has written the code. That makes a dictionary attack against the encryption feasible. All those objections may be met by using an *obfuscating compiler* to generate the FxA machine code.

We have such a C compiler written in Haskell at http://nbd.it.uc3m.es/~ptb/obfusc_comp-0_9.hs. Now understand by *location* a register or memory cell, then:

Fact 2. *There is a strategy for compiling to FxA code such that the probability across different compilations that any particular runtime 32-bit value x for $[x]_{\mathscr{E}}$ is in location l at any given point in the program is uniformly $1/2^{32}$.*

'The (obfuscating) compiler did it' is a valid cover story for any runtime cipherspace collision. The compiler's strategy takes advantage of the constants in each FxA instruction to vary the runtime data written at location l by a random offset each time the same source code is (re-)compiled.[??]

For example, the paradigmatic Ackermann function [44]

```
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));
}
```

(this function has as much computational complexity as any computable function) compiles to FxA code that runs with the trace shown in Table 8 for arguments (3,1). Although the source contains only the constants 0, 1, the trace shows that the FxA instructions have been compiled with seemingly random embedded constants (the decrypted form is shown in the table, with an 'E[-]' to indicate encryption). The trace

TABLE 8. RUNTIME TRACE (ABRIDGED) FOR THE ACKERMANN FUNCTION ON (3,1), RESULT 13.

```
PC  instruction                                    update
 ...
35  add t0 a0  zer E[-86921031]       t0 = E[-86921028]
36  add t1 zer zer E[-327157853]      t1 = E[-327157853]
37  beq t0 t1  2   E[240236822]
38  add t0 zer zer E[-1242455113]     t0 = E[-1242455113]
39  b 1
41  add t1 zer zer E[-1902505258]     t1 = E[-1902505258]
42  xor t0 t0  t1  E[-1734761313] E[1242455113] E[1902505258]
                                      t0 = E[-17347613130]
43  beq t0 zer 9   E[-1734761313]
53  add sp sp  zer E[800875856]       sp = E[1687471183]
54  add t0 a1  zer E[-915514235]      t0 = E[-915514234]
55  add t1 zer zer E[-1175411995]     t1 = E[-1175411995]
56  beq t0 t1  2   E[259897760]
57  add t0 zer zer E[11161509]        t0 = E[11161509]
 ...
143 add v0 t0  zer E[42611675]        v0 = E[13]
 ...
147 jr  ra
STOP
```

Legend: E[-] denotes an encrypted value. Instructions as in Table 7.

shows random (and encrypted) runtime data values being written to registers before the return value of (encrypted) 13 is written. That is only recognisable because the compiler has been told on this occasion not to vary the result value in register **v0** (the standard OpenRISC return value register). Otherwise the return value would have been another random-looking value, displaced, however, by an amount known to the code author.

Every time the compiled code is to be reused on new (encrypted) data, it must be recompiled and reencrypted from source, so the compiler can introduce new variations. But if the input data is known in advance, it can safely be embedded in the code and the code furnished with an outer loop over the data instances. So, the platform is primarily practical as a secure server for arbitrary remote computations with that mode of working. However, it seems that the obfuscating scheme may be varied dynamically at runtime too, though we have not yet begun to explore the details, and that would allow continuous computation to be carried out safely.

Whatever the ultimate mode of use, Fact 2 formally implies[??] the cryptographic notion of *semantic security* for runtime data against the operator as adversary. That is, the data is as safe as if the computation were carried out in a locked safe, only the final output showing [45].

It is planned to equip the prototype processor for FxA by installing an extra pre-decode stage to split incoming FxA instructions into sequences of OpenRISC instructions.

7. Conclusion

Computing in which user data is secure against the operator, operating system and other insiders is logically possible, using the 'encrypted computing' approach. That means a processor that 'works encrypted', taking encrypted inputs and passing them through encrypted intermediate states to produce encrypted outputs, plus the appropriate machine code instruction set, plus an 'obfuscating compiler'. Each of those is logically necessary for security, and the three together can be proved to provide it. The question is whether the combination forms a system that works well and quickly enough to be acceptable or practical. This paper has attempted to show that the answer is already 'yes', in order to interest the community in bringing the hardware to state of the art levels.

A superscalar pipelined design for a 32-bit RISC processor that works encrypted has been described here as the basis of that system. It uses the principle of a modified arithmetic to generate encrypted working. Performance measures running the standard OpenRISC instruction set with AES encryption have been reported, achieving around the level of a 330 MHz Pentium M, given a 1 GHz clock. We believe that these figures should be enough to provide a convincing case.

In the processor, the operator has unlimited, privileged access to registers and memory in the conventional way, yet cannot access unencrypted user data, because of the hardware protocol embedded in the design. The 'FxA' modified RISC instruction set has the property that the encrypted data in every program code and trace may be interpreted arbitrarily, allowing semantic security to be proved for data in the three-part system of processor, instruction set and compiler.

References

[1] P. Breuer and J. Bowen, "A Fully Homomorphic Crypto-Processor Design: Correctness of a Secret Computer," in *Proc. Int. Symp. Eng. Sec. Soft. Syst. (ESSoS'13)*, no. 7781 in LNCS, pp. 123–138, Springer, Feb. 2013.

[2] P. Breuer, J. Bowen, E. Palomar, and Z. Liu, "On obfuscating compilation for encrypted computing," in *Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17)* (P. Samarati, M. Obaidat, and E. Cabello, eds.), vol. 6, pp. 247–254, SCITEPRESS, July 2017.

[3] M. Barr, "Memory," in *Programming Embedded Systems in C and C++* (A. Oram, ed.), ch. 6, pp. 64–92, O'Reilly Media, Inc., 1st ed., 1998.

[4] P. Breuer and J. Bowen, "Avoiding Hardware Aliasing: Verifying RISC Machine and Assembly Code for Encrypted Computing," in *Proc. 2nd IEEE Work. Rel. Sec. Data Anal. (RSDA'14), IEEE Int. Symp. Soft. Rel. Eng. Work. (ISSREW) 2014*, pp. 365–370, IEEE, Nov. 2014.

[5] D. A. Patterson, "Reduced instruction set computers," *Commun. ACM*, vol. 28, pp. 8–21, Jan. 1985.

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

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

[8] P. Breuer and J. Bowen, "A Fully Encrypted Microprocessor: The Secret Computer is Nearly Here," in *Proc. 7th Int. Conf. Amb. Syst., Netw. Tech. (ANT'16) & Affil. Work.* (E. Shakshuki, ed.), vol. 83 of *Procedia Comp. Sci.*, pp. 1282–1287, Elsevier, Apr. 2016.

[9] P. Breuer, J. Bowen, E. Palomar, and Z. Liu, "A Practical Encrypted Microprocessor," in *Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16)* (C. Callegari, P. Sarigiannidis, *et al.*, eds.), vol. 4, pp. 239–250, SCITEPRESS, July 2016.

[10] S. Rass and P. Schartner, "On the security of a universal cryptocomputer: The chosen instruction attack," *IEEE Access*, vol. 4, pp. 7874–7882, 2016.

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

[12] L. Knudsen, V. Rijmen, R. Rivest, and M. Robshaw, "On the design and security of RC2," in *Proc. 5th Int. Work. Fast Soft. Encr. (FSE'98)* (S. Vaudenay, ed.), pp. 206–221, Springer, Mar. 1998.

[13] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Conf. Th. App. Crypt. Tech. (EuroCRYPT'99)*, no. 1592 in LNCS, pp. 223–238, Springer, 1999.

[14] Intel Corp., "Intel 64 and IA-32 architectures optimization reference manual," 2016. Ch. C: Instruction Latency and Thruput.

[15] N. Tsoutsos and M. Maniatakos, "The HEROIC framework: Encrypted computation without shared keys," *IEEE Trans. CAD IC Syst.*, vol. 34, no. 6, pp. 875–888, 2015.

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

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

[18] M. Buer, "CMOS-based stateless hardware security module," Apr. 6 2006. US Pat. App. 11/159,669.

[19] D. Hardin, "Real-time objects on the bare metal: An efficient hardware realization of the JavaTM virtual machine," in *Proc. 4th Int. Symp. Obj.-Oriented Real-Time Dist. Comp. (ISORC'01)*, pp. 53–59, IEEE, 2001.

[20] M. Schoeberl, "Java technology in an FPGA," in *Proc. 14th Int. Conf. Field-Prog. Logic App. (FPL'04)* (J. Becker, M. Platzner, *et al.*, eds.), pp. 917–921, Springer, Aug. 2004.

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

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

[23] R. Ostrovsky and O. Goldreich, "Comprehensive software protection system," June 16 1992. US Patent 5,123,045.

[24] S. Lu and R. Ostrovsky, "Distributed oblivious RAM for secure two-party computation," in *Proc. Th. Crypto.*, pp. 377–396, Springer, 2013.

[25] R. Hartman, "System for seamless processing of encrypted and non-encrypted data and instructions," 1993. US Pat. 5224166.

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

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

[28] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. 19th Ann. Int. Crypto. Conf. (CRYPTO'99)* (M. Wiener, ed.), Advances Crypto., pp. 388–397, Springer, 1999.

[29] D. Patterson and J. Hennessy, "MIPS R2000 assembly language," in *Computer Organization and Design: the Hardware/Software Interface*, ch. A.10, Morgan Kaufmann, 1994.

[30] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. 2nd Int. Work. H'w. Arch. Supp. Sec. Priv. (HASP'13)*, ACM, June 2013.

[31] F. Schuster *et al.*, "VC3: Trustworthy data analytics in the cloud using SGX," in *IEEE Symp. Sec. Priv.*, pp. 38–54, May 2015.

[32] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *Proc. 10th Eur. Work. Sys. Sec. (EuroSec'17)*, pp. 2:1–2:6, ACM, 2017.

[33] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *Proc. 30th Ann. Int. Conf. Th. App. Crypto. Tech. (EuroCRYPT'11)*, no. 6632 in LNCS, pp. 129–148, Springer, 2011.

[34] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Adv. Crypto. – Proc. 33rd Ann. Crypto. Conf. (CRYPTO'13)* (R. Canetti and J. Garay, eds.), no. 8042 in LNCS, pp. 75–92, Springer, Aug. 18-22 2013.

[35] K. Lauter, M. Naehrig, and V. Vaikuntanathan, "Can homomorphic encryption be practical?," in *Proc. 3rd ACM Cloud Comput. Sec. Work. (CCSW'11)*, pp. 113–124, ACM, 2011.

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

[37] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proc. 2nd Ann. Comp. Sec. App. Conf. (ACSAC'06)*, pp. 473–482, IEEE, 2006.

[38] M. Maas *et al.*, "Phantom: Practical oblivious computation in a secure processor," in *Proc. ACM Conf. Comp. Commun. Sec. (SIGSAC'13)*, pp. 311–324, ACM, 2013.

[39] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *Proc. Int. Conf. Arch. Supp. Prog. Lang. Op. Syst. (ASPLOS'15)*, 2015.

[40] P. Simmons, "Security through amnesia: A software-based solution to the cold boot attack on disk encryption," in *Proc. 27th Ann. Comp. Sec. App. Conf. (ACSAC'11)*, pp. 73–82, ACM, 2011.

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

[42] J. A. Halderman *et al.*, "Lest we remember: cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[43] K. Hwang, *Advanced Computer Architecture*. Comp. Sci., Tata McGraw-Hill Education, 2011. 2nd ed.

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

[45] S. Hada, "Zero-knowledge and code obfuscation," in *Proc. 6th Int. Conf. Th. App. Crypt. Info. Sec. (ASIACRYPT'00)* (T. Okamoto, ed.), no. 1976 in LNCS, pp. 443–457, Springer, 2000.