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 in a processor in theory causes user data to automatically remain in encrypted form throughout processing, providing a new technical approach to privacy against other users and the operator that is initially intended for use in cloud-based computation. The theory has now been checked in a prototype superscalar processor design that conforms to the OpenRISC specification. The processor runs in a (one-to-many) encryption in user mode, and runs unencrypted in supervisor mode. What architectural features work well and what does not work well in the processor architecture is reported here, mapped via performance measures. We are reporting cycle-accurate measurements on models that embed AES-128 (symmetric, keyed; the US 'advanced encryption standard') and Paillier-72 (asymmetric, additively homomorphic, no key in-processor) encryptions in a 128-bit architecture, and RC2-64 encryption (symmetric, keyed) in a 64-bit architecture. 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 experience gained allows us to propose here a RISC instruction set modified for encrypted working, with which the cryptographic notion of semantic security for user data is formally guaranteed on this kind of platform, in cooperation with an 'obfuscating' compiler. But the aim of this paper is to raise awareness that computing provably secure against insiders is not only logically achievable this way but plausibly practical and marketable, encouraging the focus of the hardware community.

i

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 is 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.[1] That is to say, secure computing in the cloud on behalf of a remote user [2] is 'formally solved' by this approach and it remains to be seen if engineering a processor design following this approach to meet practical expectations is possible.

ii 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 community to apply the approach to encrypted comput-
iii ing with more state-of-the-art computer architectures. The authors are not specialist computer or hardware engineers, but computer scientists, mathematicians, security experts, cryptographers, and software engineers, and we believe specialists will do much better there than we.

Another objective of this paper is to convince the community that it does all work. That this class of processor works via a changed arithmetic ought to seem counter-intuitive to an engineer, on the basis that a tiny bug in a computer program or hardware unit gives rise to wrong program results, so changing *all* the arithmetic at once should be catastrophic. Further, with a one-to-many encryption, the encryption of a given value varies, so, an encrypted memory address may change during processing and miss the intended target in RAM. The effect appears as 'hardware aliasing' [3] to software, and how to compile for that is documented [4] (reuse encrypted addresses, not recalculate them, as the latter reencrypts the address differently).

Since conventional instruction sets are insecure in an encrypted computing context (see paragraph 2.6), we also set out here a modified machine code instruction set architecture that fixes the deficiencies. Encrypted computing cannot be secure without it. But it also requires backing by a form of randomised compilation, or encryption would intrinsically 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 sketched here.

iv In the rest of this section, general points, particularly those raised by the referees, are dealt with before the organisation of the paper is set out. Margin numerals are rebuttal notes.

1.1 **Coverage.** The 'bells and whistles' of operating system support in the processor are now known to be generally compatible with encrypted running. That has been proved by satisfying the whole of the OpenRISC v1.1 specification at `http://openrisc.io`, with minor restrictions for secu-

[1] An appendix supplies sketches of this and other formal results to avoid breaking anonymity by referencing the refereed publications.

A superscript capital letter in the text refers to an appendix section. In camera ready copy, these references would be replaced by citations. The particular claim footnoted here is explained in Section 6.

rity purposes.[2] Both user and supervisor modes are implemented. Operating system code runs (unencrypted) in supervisor mode on the platform and a successful, working micro operating system supports the user-mode tests reported here.

1.2 **Security.** An engineer should always project a natural scepticism at any claim of security, because of oversights in design and implementation that occur. But that is exactly the point, because the security in this approach depends fundamentally on mathematics, abetted by but not based on engineering. Engineering is needed to get it right and get it fast, but is not responsible for the mechanism of protection. Vulnerabilities could arise via mistakes of implementation, but not unknown weaknesses in the principles. The effectiveness of the hardware protocols in keeping supervisor mode from accessing in unencrypted form any datum that has been
v generated in user mode is proved.[A] With that as basis, in conjunction with the slightly modified instruction set described here and a special compiler,[C] the cryptographic notion of semantic security [5] (no attack does better than blind guess-
vi work) has been proved to hold for user data in computations on the platform,[D] which leaves no room to doubt security.

vii A security expert who is not familiar with computer architecture should be on notice that it is physically impossible for hardware to make digitally available internal data that there are no hardware traces to read. In particular, the content of the processor pipeline at any moment is unavailable. Encryption is done in hardware as part of the pipeline, using internal pipeline registers only, and always takes the same time (and power) so timing attacks [6] and cache attacks [7] are not applicable. A potential exception to the visibility of half-completed or aborted pipeline instructions is via subsequent cache behaviour and timing, but the user caches are not available to the supervisor mode of the processor, nor are user branch prediction and cache hit statistics. These and other subtleties of security engineering are discussed in Section 4 and hardware engineers should feel free to add to our solutions, as defenses for conventional processors apply.

The security expert should also note that there is no intent to hide control flow. The operator is free to observe and to guess what the (encrypted) program running might be. It is the numerical value of runtime user data under the encryption that can formally be shown to be secure. An observer might guess, for example, that an image analysis program is running, but should be unable to deduce the coordinates of the rocket launching site detected by the analysis.

To cheaply secure the control flow too, the user can run the code for a virtual machine (VM) on the platform, and run their computation on that. The encrypted state of the VM, including its control state (program counter, etc.), is encrypted data that is transformed by the same arithmetic each cycle, and the security of that is guaranteed.

More efficiently, the user may effectively obfuscate their program for the platform by compiling it so that both branches of every **if** statement are executed, but they are combined arithmetically at the join so that only one branch contributes, via statements of the form $x \leftarrow a*x_1 + b*x_2$, where x_1, x_2 are computed down the two branches and a, b are encrypted, calculated, 0/1 values. Arithmetic is encrypted on the platform, so the program trace looks the same (up to length) to an observer whatever the inputs, which is the definition of cryptographic obfuscation [8]. It is just beginning to be realised that cryptographic obfuscation is both possible and practical with help from hardware [9].

Guarantees of data security hold on this platform even when a program tries to give secrets away, for example by looping three times when returning '3', encrypted. The mathematics shows that an observer cannot be sure that it is '3',[B]. If the program code is compiled to avoid numerical biases, then any other value is formally equally as likely beneath the encryption.[D] Section 6 gives an example trace for a simple compiled program that should make these statements more comfortable for the general reader.

The cycle-accurate model processors reported here have been built to (i) confirm that those theoretical principles work and do so usably well in practice, and (ii) gain experience and explore the practical limits of this approach.

1.3 **Usability.** With respect to (i) above, the models have provided good metrics and significant measures are reported in Section 5. The single pipeline emits more than one instruction every two clock pulses for standard application code mixes. The measured speed on standard benchmarks using a 1 GHz base clock is that of a 433–582 MHz classic Pentium (differences between compilation for the RISC instruction set here and Intel's CISC instruction set should be taken into account). Concern for performance has improved the architecture to work measurably well, despite natural obstacles that mean encrypted running will be slower. In particular, encrypted code is longer, and sub-word operations (e.g., read a byte, write a byte) must be carried out via multiple arithmetic operations on full words. The practical question is how much slower is encrypted computing, because numbers help future adopters or partners decide on options. The instruction test numbers reported here are 50% better than the numbers in [10], where the architectural framework that we have taken forward is published. But benchmark results are available for the first time and are 7× better than for the original (104–140 MIPS instead of 20 MIPS; see Section 5), albeit partly due to improved compilation.

1.4 **Limits.** With respect to (ii) above, it was originally unknown if any conventional instruction set could be completely compatible with encrypted running. Now that the situation is better understood, it is clear that not *every* program can run encrypted – compilers and programs that transform the addresses of program instructions (as distinct from addresses of program data) must run unencrypted because program addresses are unencrypted by design – that prevents 'known plaintext attacks' (see, e.g., [11]) on encrypted address sequences. But within those limits applications run well in the modified OpenRISC instruction set and the necessary system code support is successfully provided by the processor. The largest application suite[3] ported so far is 22,000 lines of C, but it and other ports (now about fifty

[2] In user mode, no register may contain a value known in advance, or a 'known plaintext attack' could be mounted against the encryption. For example, the register that should contain the processor version number may not return it (encrypted) in user mode.

[3] IEEE floating point test suite at `http://www.jhauser.us/`
viii `arithmetic/TestFloat.html`. Floating point arithmetic to the IEEE specification has previously been seen as impractical for encrypted computing, leading to alternative proposals [12].

programs) have worked well, surprising these authors. We expect programs with memory footprints that do not fit in cache (too large for practical simulation) to perform badly encrypted, but we also expect the problem to become moot by virtue of external factors (processor cache may be a lot bigger in future) or for it to be overcome. The root of the problem is that unit granularity is required in address logical to physical translations, as encrypted addresses do not cluster in pages, and that is very inefficient. However, the processor has internal access to the decrypted address, and we are experimenting with that in order to recover performance with security.

ix 1.5 **Simulation.** The question of if our simulations are valid deserves a preliminary answer, as the hardware community may be unfamiliar with the use for model-driven design and exploration [13], rather than validating an existing design.

In the first place, it does not matter if simulations are accurate, as our results are for encrypted as opposed to unencrypted working in the same basic design (50-80% as fast), which is also the approach taken in the accounts of other contemporary designs reported in Section 3. If there are mistakes, they are the same mistakes on either side of the division sign. Secondly, mistakes that make a design appear faster than it should be would make the comparison worse, as the slowness of encrypted working over unencrypted working becomes more exposed. Thirdly, the design is clocked at less than a quarter of the speed fabricated chips run at nowadays, so it is not pushing the envelope. Care has been taken to configure conservative latencies for memory and cache (13.5ns and 3ns respectively, adjustable). Fourthly, there is no room to get a pipeline simulation wrong in software – the pipeline moves instructions on by one pipeline stage at every clock tick. Instructions that need data do not move on until it is available, creating an empty space in the pipeline ahead and 'stalling' the instructions behind. The simulation would not work at all if that were miscoded, and timing is counting clock ticks. Model development has been incremental on that foundation, and we have been able to detect anomalies along the way. More than a million lines of final code are invested in the simulation software, for an estimated 25 years of total software engineering effort. The difficult code relates to the interaction between memory, cache and pipeline timings, but only programs with footprint that fits in the (8 MB, write-back) cache are reported here, and they are bug-free in the respect that data is always written before it is read. That ensures a cache hit on read. So, it suffices that cache interactions alone are correctly modelled in order for the results to be correct. RAM modelling is not a practical issue for dispute. Fifth, running unencrypted at 1GHz, the machine is registering 199 MIPS for the (unoptimised) Dhrystones v2.1 benchmark, or 0.2 MIPS/MHz. That is only half as good as ARM RISC machines a decade old, so the numbers from simulation pass sanity checking. Sixth and lastly, simulation is specification, not validation. It identifies what engineers working at the logic gate level must achieve. A gate-level simulation on silicon (aka 'reality') will have its own numbers. Simulation drives what becomes real, not preempts it.

x 1.6 **What is new?** Conventional benchmark results for this class of architecture are reported for the first time, which has required development of a significant software infrastructure in support. Measures carried over from earlier reports have been bettered by 50% and it is detailed here what architectural optimisations work to make that happen. Correct interaction with operating system support has been experimentally validated now, as well as proved secure, which boosts confidence in the practicality of the approach. This is also the first report of the approach for a computer engineering audience. We are also able here to pair it for the first time with theory that gives formal foundations for security. For that, a specially modified instruction set architecture is crucially necessary and it is shown here for the first time. Security in encrypted computing cannot work without it, so it does engineers no good to ignore it.

1.7 **Organisation.** The organisation of this article is as follows. Section 2 summarises processor design and working in this approach to encrypted computing, giving bullet points for the reader to take away. The reader should look there first for an understanding. This approach does not work via a radically different architecture so a detailed schematic is
xi not included here. The design is as expected for the decades-old classic single pipeline RISC processor [14], with modifications as described. The 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
xii are achieving (of the order of 1 logical gate operation per second) in their experiments using homomorphic encryption [15, 16] as their approach to encrypted computing (homomorphic encryption gives rise to a homomorphic arithmetic, which is the 'appropriately modified' arithmetic described in the first sentence of this Introduction).

The security engineering considerations arising from putting these principles into practice are described in Section 4. A reader should look there for the minutia of the hardware concerns, but more is discussed. It should be noted in particular that key management is not in any way a concern here, both by virtue of the security arguments given and because it can be done via extant hardware.

Performance is taken up in Section 5, which comprises the heart of the paper. An 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 pipelining is unsuccessful in allowing one instruction to start while another is still running. The reason is that the 'execution' stage of an arithmetic instruction necessarily occupies nearly the whole of the pipeline, but the results cannot be anticipated, so a following instruction must generally wait to start for the one ahead to complete, making the pipeline one-instruction-at-a-time.

The security vulnerabilities of conventional instruction sets were explained in Section 2, but Section 6 shows a modified instruction set that makes encrypted computing provably secure. It requires, however, code to have been compiled by an 'obfuscating' compiler that takes advantage of the possibilities in the instruction set. The requirements for the compiler
xiii are sketched in that section (the compiler's URL is given). Pro-

cessor, instruction set and compiler together comprise a three-point system for provable security in encrypted computing.

2. REFERENCE POINTS

This section summarises the basic working of the processor architecture framework introduced in [10] and intends to provide touchstone points to which the reader can refer.

2.1 **Architecture.** The processor architecture is based on the classic single pipelined RISC design of [14], clocked here at a nominal 1 GHz with 3 ns cache. Register layout and functionality follows the OpenRISC v1.1 architectural specification at `http://openrisc.io`. There are 32 general purpose registers (GPRs) and up to 65,536 special purpose registers (SPRs). Some SPRs' 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), in both cases holding either encrypted 32-bit data or unencrypted 64-bit data.

The pipeline incorporates standard 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.

2.2 **Modes.** There are just two processor modes: *user* and *supervisor* mode, as per the OpenRISC specification. User mode works encrypted on data that is 32-bit beneath the encryption and supervisor 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.

2.3 **Adversaries.** The operator is the notional adversary who tries to read the user's data, and/or rewrite it. By 'the operator' is meant also the supervisor mode of operation of the processor, in which instructions have access to every register and every memory location. As the most privileged user on the machine, the operator stands in for all as adversary. If user data is secure from the operator, it is secured from all.

2.4 **Simulation.** The open source OpenRISC 'Or1ksim' simulator, available from `opencores.org/or1k/Or1ksim`, was the original source of the simulation code, so the OpenRISC instruction semantics were initially correct. It has been mod-
xiv ified to a cycle-accurate pipeline simulator, in which the original monolithic instruction semantics for OpenRISC have been split into decode, prep, read, execute, post and write steps for the pipeline (there are multiple prep and post steps, dedicated to the encrypted arithmetic). The decode stage code is generated from specifications, as in the original source. The decode stage creates a 'microcode' that accompanies the instruction down the pipeline and configures the processing of the instruction in each later pipeline stage. Some bits of the microcode are writable, allowing the semantics to vary dynamically. If data is available from ahead in the pipeline before it is available from registers is marked in the microcode, for example. The open source code archive and history is at `http://sf.net/p/or1ksim64kpu`.

2.5 **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 16-bit immediate data field of the OpenRISC standard, 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 modifications are minor. The displacement (as in an array) from the base address embedded in a load or store
xv instruction must be zero in user mode, in order to reduce the amount of arithmetic required in the pipeline. The displacement register for the SPR index is ignored in a 'move to/from SPR' instruction in user mode because it would be a give-away for the encryption. A shift instruction has its fields reordered so the shift is contiguous. That is all.

xvi The OpenRISC 32-bit floating point as well as integer instructions have been fully implemented for user mode.

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

2.6 **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 possible to write (unencrypted, supervisor 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 for the system support provided so far (e.g., memory fault handlers, I/O routines). But the experience has clarified that standard instruction sets are inherently insecure [17] with respect to the operator as adversary, who may steal an (encrypted) user datum x and put it through the machine's division instruction to get x/x, which is an encrypted 1. With a 1 in hand, it is simple to construct any encrypted y by repeatedly applying the machine's addition instruction. By comparing the encrypted $1, 2, 4,$ etc. so-obtained with any encrypted z using the instruction set's comparator instructions (testing $2^{31} \le z$, $2^{30} \le z$, $\dots$ 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 even statistically right at a level above chance. The FxA instructions are described in Section 6.

2.7 **Encryption.** The processor models have been tested while embedding Rijndael-64 and -128 symmetric ciphers (the latter is the US Advanced Encryption Standard (AES) [18]), RC2-64 [19] and Paillier-72 [20]. The last is an additively homomorphic[4] 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 associated hardware fits in the pipeline, taking up most of the 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. The complete AES encryption/decryption unit occupies 10 stages. For other symmetric ciphers, between 1 and 20 pipeline stages can be configured in the simulator. For homomorphic

[4]'Homomorphic' for Paillier means multiplication of encrypted numbers corresponds to addition of the unencrypted numbers.

encryptions a multistage arithmetic unit occupies their place.

Note that the encryptions here, 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, xvii allowing a trace to be audited. For Paillier, 'blinding' multipliers are generated instead.[6]

The choice of encryptions has been dictated by the development path. The Or1ksim simulator was 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 [21, 22] (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).

2.8 **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. xviii Executables are written as standard ELF format files. An experimental 'obfuscating' compiler, as described in Section 6, is also coming on-line. We are also experimenting with the appropriate format for read-only global data in ELF files. Each (encrypted) word must have its unique (encrypted) address specified along with it, but it is unsatisfactory to dedicate an ELF section to each. Yet that is the only fully standards-compatible way to do it in ELF. A custom solution would require matching changes in program loaders. Global variables are a point of difficulty for the compiler too, 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.

2.9 **Limits.** Word widths up to 2048 bits are contemplated for current technology. Memory paths would have to be appropriately broadened from the current 64 bits, and memory accesses paralleled correspondingly.

2.10 **Configuration.** Tests are centered about a 15-stage pipeline configuration with a nominal 1 GHz clock. Of that, 10 stages are for the modified arithmetic, 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 a full word (64 or 128 bits or more) wide, the program memory path is still a legacy 64 bits wide, which results in two 32-bit instructions per cycle being pulled to a split look-ahead/behind buffer in xix the pipeline fetch stage. The fetch stage should look at groups of instructions together because some instructions may carry extra data in one or more prefix instructions. The split fetch buffer is configurable (32 instructions wide by default), half (ahead) to span the average interval between branches in a typical code mix and half (behind) to cover for tight loops. xx The latter consideration avoids buffer thrashing. The relatively narrow (64-bit) instruction path is a continuing bottleneck that we cannot configure away because of legacy issues in the simulator. The fetch buffer only helps make up for that.

2.11 **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 [23] or introduced via a Diffie-Hellman circuit [24] or equivalent that loads the key in public view without revealing it.[5]

Hopefully, the bases have now been covered for the reader.

3. RELATED PLATFORMS

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

3.1 **HEROIC** [21] is the most comparable contemporary. It is a 16-bit processor working in Paillier-2048 encryption [20] 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 [25, 26] 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

[5]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.

unencrypted numbers x, y:

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

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.[6] 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** [27] 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' [28–30].

The idea of physical isolation plus encrypted memory has emerged many times over the years (e.g., [31, 32]) and success means doing it as well as Ascend does. Otherwise channels such as cache-hit statistics [33] and power drain [34] can give away information to a privileged observer. Ascend runs RISC MIPS instructions [35] and slows down by a factor of 12-13.5$\times$ in encrypted mode with AES-128 (absolute speeds are not given in [27]), as compared to 10-50% slowdown for our models (Section 5).

3.3 **Intel's SGX**$^{\text{TM}}$ ('Software Guard eXtensions') processor technology [36] 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 [31, 32], not encrypted computing. RAM is a peripheral to a processor.

SGX machines are used [37] 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,
xxi e.g. via timing variations and power use [33], and there have recently been successful attacks against SGX based on timing [38]. 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 [16] 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 [39] 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 [40] 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 the same as that which HEROIC overcomes: an encrypted comparison
xxii 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.

xxiii 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. [41] 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' [42] 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 supervisor-mode caches in our models, and statistics are not available to the other mode, so side-channel attacks based on cache-hits [33, 43] are not available.

3.6 **Oblivious RAM** [28–30] and its evolutions [44, 45]) is often cited as a defense against dynamic memory snooping. That is in contrast to static snooping, so-called 'cold boot' attacks [46–48] – 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.

[6] Paillier may embed random 'blinding factors' into encrypted numbers. Those are multipliers $r^n \bmod m$, 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 $r^{\phi n} = (1{+}kn)^n{=}1{+}kn^2{+}\ldots{=}1 \bmod n^2$ and does not affect the decrypted value. HEROIC's one-to-one encryption does not use different blinding factors.

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 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.^{2}

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 supervisor mode via a hardware protocol (*) sketched below.

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

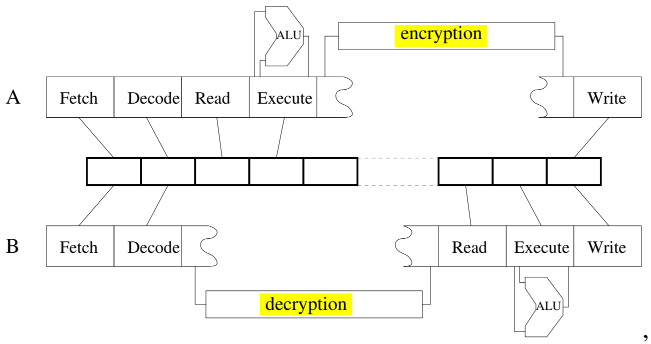

Figure 1: The pipeline is configured in two different ways, 'A' and 'B', for two different kinds of user mode instructions during encrypted working, in order to reduce encryption/decryption to at most once per instruction (this illustration of the principle is adapted from [10]).

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 supervisor 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 [10]; note that by 'arithmetic operations' is meant the arithmetic stages of individual in-

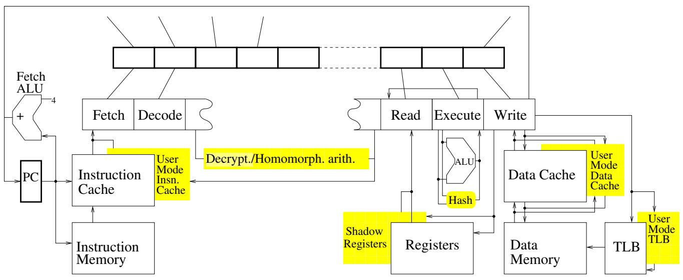

Figure 2: Pipeline integration, showing shadow units for user mode. This illustration adapts drawings in [10].

structions, not the whole instructions). In support:

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

The user mode registers 'shadow' the supervisor mode registers, as illustrated in Fig. 2, where the shadow units are shown slightly behind the non-shadow (i.e., supervisor mode) units. Where the supervisor 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 supervisor mode from accessing data unencrypted that originates in user mode, and vice versa.[A] The protocol also assures user data is stored encrypted in
xxv 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[5] 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 [10]. 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 its 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 [49]. This baseline is 4% better than the numbers reported in [10]. The test spends 52.7% of its time in user mode, as against 54.8% in [10] (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

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%)

multiplication, which is entirely done in software. Table 3 compares RC2 with Paillier on the 'multiplication test':

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 RC2 (64-bit) cycles		forwarding ✓	forwarding ×
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[7]; (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
xxvi shown in the first group of three lines in Table 6:

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 MHz	ARM250 12	ARM926 200	ARM1176 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

[7]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.

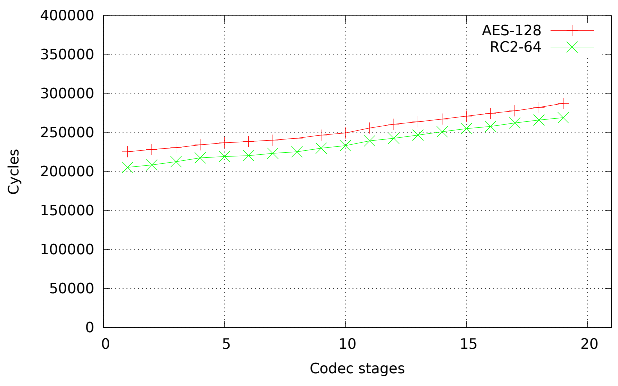

Figure 3: Number of executed cycles with symmetric encryption for the 'add test' of Table 1 against number of stages occupied by the multistage encryption/decryption unit ('codec'), showing 2.5% extra cost per stage.

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.
xxvii For comparison, the benchmark run in supervisor 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 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.

The Intel/AMD 'ASENC' AES round instruction has latency 0.95ns (4 cycles at 4.2GHz) on Skylake cores (Table C-9 of [50]), so modelling it as one pipeline stage at 1 GHz is realistic. In any case, performance results may be extrapolated as needed: Fig. 3 shows each extra pipeline stage costs
xxviii 2.5%, both with AES and RC2 encryptions.

6. FXA INSTRUCTION SET

Standard instruction sets are insecure for encrypted working (recall the argument in 2.6 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.

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.

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:

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

xxix 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 pseudorandom 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[B] 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 en-

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.

cryption 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.[C]

For example, the paradigmatic Ackermann function [51]

```
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 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[D] 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 [5].

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.

xxx 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.

8. REFERENCES

[1] P. T. Breuer and J. P. Bowen, “A Fully Homomorphic Crypto-Processor Design: Correctness of a Secret Computer,” in *Proc. Int. Symp. Eng. Sec. Softw. Syst. (ESSoS'13)*, no. 7781 in LNCS, (Heidelberg/Berlin), pp. 123–138, Springer, Feb. 2013.

[2] M. van Dijk and A. Juels, “On the impossibility of cryptography alone for privacy-preserving cloud computing.,” *HotSec*, vol. 10, pp. 1–8, 2010.

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

[4] P. T. Breuer and J. P. Bowen, “Avoiding Hardware Aliasing: Verifying RISC Machine and Assembly Code for Encrypted Computing,” in *Proc. 2nd IEEE Worksh. Rel. Sec. Data Anal. (RSDA'14), IEEE Int. Symp. Softw. Rel. Eng. Worksh. (ISSREW) 2014*, pp. 365–370, IEEE, Nov. 2014.

[5] S. Hada, “Zero-knowledge and code obfuscation,” in *Proc. 6th Int. Conf. Theor. Applic. Crypt. Inform. Sec. (ASIACRYPT'00)* (T. Okamoto, ed.), no. 1976 in LNCS, pp. 443–457, Springer, 2000.

[6] P. C. Kocher, “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,” in *Proc. 16th Ann. Int. Crypto. Conf. (CRYPTO'96)* (N. Koblitz, ed.), no. 1109 in LNCS, (Berlin/Heidelberg), pp. 104–113, Springer, Aug. 1996.

[7] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of AES,” in *Proc. Cryptographers' Track at RSA Conf. (CT-RSA'06)* (D. Pointcheval, ed.), no. 3860 in LNCS, (Berlin/Heidelberg), pp. 1–20, Springer, Feb. 2006.

[8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, “On the (im) possibility of obfuscating programs,” in *Proc. 21st Ann. Int. Crypto. Conf.* (J. Kilian, ed.), no. 2139 in LNCS, pp. 1–18, Springer, Aug. 2001.

[9] K. Nayak, C. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal, “HOP: Hardware makes obfuscation practical,” in *Proc. 24th Ann. Net. Dist. Sys. Sec. Symp. (NDSS'17)*, 2017.

[10] P. T. Breuer and J. P. Bowen, “A Fully Encrypted Microprocessor: The Secret Computer is Nearly Here,” *Procedia Comp. Sci.*, vol. 83, pp. 1282–1287, Apr. 2016.

[11] E. Biham and P. C. Kocher, “A known plaintext attack on the pkzip stream cipher,” in *Proc. 2nd Int. Worksh. Fast Softw. Encryption (FSE'94)* (B. Preneel, ed.), no. 1008 in LNCS, pp. 144–153, Springer, Dec. 1994.

[12] V. Dimitrov, L. Kerik, T. Krips, J. Randmets, and J. Willemson, “Alternative implementations of secure real numbers,” in *Proc. ACM SIGSAC Conf. Comp. Comm. Sec. (CCS'16)*, (New York, NY, USA), pp. 553–564, ACM, 2016.

[13] H. Giese, G. Karsai, E. A. Lee, B. Rumpe, and B. Schätz, eds., *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers*, vol. 6100 of *LNCS*. Berlin/Heidelberg: Springer, 2010.

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

[15] 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.

[16] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proc. 41st Annu. Symp. Theor. Comp. (STOC'09)*, pp. 169–178, ACM, 2009.

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

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

[19] L. R. Knudsen, V. Rijmen, R. L. Rivest, and M. J. B. Robshaw, “On the design and security of RC2,” in *Proc. 5th Int. Worksh. Fast Softw. Encr. (FSE'98)* (S. Vaudenay, ed.), pp. 206–221, Springer, Mar. 1998.

[20] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Conf. Theor. Applic. Crypt. Tech. (EUROCRYPT'99)*, no. 1592 in LNCS, pp. 223–238, Springer, 1999.

[21] N. G. Tsoutsos and M. Maniatakos, “The HEROIC framework: Encrypted computation without shared keys,” *IEEE Trans. CAD Integ. Circ. & Syst.*, vol. 34, no. 6, pp. 875–888, 2015.

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

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

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

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

[26] M. Schoeberl, “Java technology in an FPGA,” in *Proc. 14th Int. Conf. Field-Programmable Logic Applic. (FPL'04)* (J. Becker, M. Platzner, and S. Vernalde, eds.), pp. 917–921, Springer, Aug. 2004.

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

[28] R. Ostrovsky, “Efficient computation on oblivious RAMs,” in *Proc. 22nd Annu. ACM symp. Theor. Comp.*, pp. 514–523, ACM, 1990.

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

[30] S. Lu and R. Ostrovsky, “Distributed oblivious RAM for secure two-party computation,” in *Proc. Theor. Cryptog.*, pp. 377–396, Heidelberg/Berlin: Springer, 2013.

[31] R. Hartman, “System for seamless processing of encrypted and non-encrypted data and instructions,” June 29 1993. US Patent 5,224,166.

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

[33] 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)*, (NY), pp. 305–316, ACM, 2012.

[34] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Proc. 19th Annu. Int. Cryptol. Conf. (CRYPTO'99)* (M. Wiener, ed.), Advances Cryptol., pp. 388–397, Heidelberg/Berlin: Springer, 1999.

[35] D. Patterson and J. Hennessy, “MIPS R2000 assembly language,” in *Computer Organization and Design: the Hardware/Software Interface*, ch. A.10, San Mateo, CA: Morgan Kaufmann, 1994.

[36] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, “Innovative technology for CPU based attestation and sealing,” in *Proc. 2nd Int. Worksh. Hardw. Arch. Supp. Sec. Priv.*, 2013.

[37] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, “VC3: Trustworthy data analytics in the cloud using SGX,” in *IEEE Symp. Sec. & Privacy*, pp. 38–54, May 2015.

[38] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, “Cache attacks on Intel SGX,” in *Proc. 10th Eur. Worksh. Sys. Sec. (EuroSec'17)*, (New York, NY, USA), pp. 2:1–2:6, ACM, 2017.

[39] C. Gentry and S. Halevi, “Implementing Gentry's fully-homomorphic encryption scheme,” in *Proc. 30th Annu. Int. Conf. Theor. & Applic. Cryptog. Tech. (EUROCRYPT'11)*, no. 6632 in LNCS, pp. 129–148, Springer, Heidelberg/Berlin, 2011.

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

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

[42] 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.

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

[44] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, “Phantom: Practical oblivious computation in a secure processor,” in *Proc. ACM Conf. Comp. Commun. Sec. (SIGSAC’13)*, (New York), pp. 311–324, ACM, 2013.

[45] 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.

[46] P. Simmons, “Security through amnesia: A software-based solution to the cold boot attack on disk encryption,” in *Proc. 27th Annu. Comp. Sec. Applic. Conf. (ACSAC’11)*, (New York), pp. 73–82, ACM, 2011.

[47] 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.

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

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

[50] Intel Corp., “Intel 64 and IA-32 architectures optimization reference manual,” June 2016. ch. C: Instruction Latency and Thruput.

[51] Y. Sundblad, “The Ackermann function. a theoretical, computational, and formula manipulative study,” *BIT Num. Math.*, vol. 11, pp. 107–119, Mar. 1971.

APPENDIX

In order to preserve the anonymity protocol, proofs of the ‘facts’ claimed in the text are sketched here rather than being referenced back to the refereed publications where they may be found, as that would be too big a give-away.

In order to get to the second proof here, a little of the obfuscating compiler strategy that it refers to has to be sketched too.

This appendix is not for publication. It is provided as a courtesy to referees, in deference to the double blind reviewing protocol (referees, please note that you should not break the protocol yourselves – in particular, googling should be avoided if there is risk of exposing who we authors are).

A. THE (*) PROTOCOL

To show that the protocol (*) of Section 4 separates user and supervisor mode data, one may show that it preserves three invariants that hold at processor startup, and which are then maintained forever. Identify five data types:

- ℹ 32-bit unencrypted data that originated as encrypted user data;
- ♒ encrypted user data occupying 64 or 128 bits;
- ☾ 32-bit plaintext data that originated in supervisor mode;
- ☽ 32-bit program addresses that originate in supervisor mode and have been tagged with 0x7fff in the top 16 available bits;
- * a placeholder that stands for pending decryption (ℹ) or encryption (♒) but physically looks like program address zero (☽).

The invariants are as follows:

1. In supervisor mode, *real/shadow registers contain types ♒/ℹ or ♒/* or */ℹ or ☾ / ☽* respectively.
2. In user mode, *real/shadow registers contain types ℹ/♒, or */♒ or ℹ/* or ☽/☾* respectively.
3. *Memory contains ♒ or ☾ or *.*

Note that 1) says that type ℹ unencrypted user data is not exposed in supervisor mode. ℹ/?, where ‘?’ stands for ‘anything’, is missing from all of the allowed combinations. Also note that 1) and 2) are mutually maintained by the protocol (*), as it swaps real/shadow registers on mode change.

Every instruction preserves those invariants in both processor modes – a design principle. User mode addition, for example, does ℹ/?+ℹ/?=ℹ/*, requiring type ℹ in both addend registers, otherwise it raises a ‘range’ exception and leaves registers as they are.

So all invariants hold throughout, and 1) states that the operator (supervisor mode) can never see the unencrypted value of a datum that originated encrypted in user mode.

B. PROOFS

PROOF (FACT 1). *First consider programs C constructed from the HEROIC (equivalent) instructions: assignments $x{\leftarrow}[y{+}k]_{\mathcal{E}}$ and branches based on a test $[x{<}K]_{\mathcal{E}}$.*

Suppose for contradiction that the operator has a method $f(T,C)=y$ of knowing that the output $[y]_{\mathcal{E}}$ of C encrypts y, having observed the trace T. Now imagine that every number has $h{\neq}0$ added to it under the encryption. The additions $y{\leftarrow}[x{+}k]_{\mathcal{E}}$ in C still make sense, adding k under the encryption to a number that is h more than it used to be to get a number that is h more than it used to be. Comparisons $[x{<}K]_{\mathcal{E}}$ in C need changing, however, because the x, which are h more than they used to be, now need to be compared with K' equal to $K{+}h$ for the program to make sense. So the branch instructions in the program must be modified to contain $[K']_{\mathcal{E}}$ instead of $[K]_{\mathcal{E}}$. To the operator, the new program code C' ‘looks the same’, $C' \sim C$, because one encrypted number is as meaningful as another without the key (by the ‘no collisions’ hypothesis, the operator cannot tell either by a new collision or lack of an old one that the K have changed), and the program trace T' looks the same up to the encrypted numbers in it, which the operator cannot read, so it looks the same, $T' \sim T$, and the method f must declare the output of C to be $f(T',C'){=}f(T,C){=}y$. But it is not $[y]_{\mathcal{E}}$ but $[y{+}h]_{\mathcal{E}}$, so the method fails. It does not exist.

Now suppose for contradiction that the operator builds a new program $C'{=}f(C)$ that returns $[y]_{\mathcal{E}}$. Then its constants $[k]_{\mathcal{E}}$ are found in C and its constants $[K]_{\mathcal{E}}$ likewise, because f has no way of arithmetically combining them (the ‘no collisions’ condition means they cannot be combined arithmetically in the processor and the operator does not have the encryption key). The first half of this proof shows the operator cannot read outputs $[y]_{\mathcal{E}}$ of C', yet knows what they are. That is a contradiction.

The proof applies with minor adaptations when arbitrary FxA arithmetic assignment instructions are considered in place of simple assignments $x{\leftarrow}[y{+}k]_{\mathcal{E}}$. Changing the constants in the instruction by h under the encryption allows the change by h under the encryption of the data entering and exiting the instruction. That is, replace every FxA instruction of the form $r_0 \leftarrow [(r_1-k_1)\,\Theta(r_2-k_2)+k_0]_{\mathcal{E}}$ with $r_0 \leftarrow [(r_1-k_1')\,\Theta(r_2-k_2')+k_0']_{\mathcal{E}}$ where $k_i' = k_i + h$, $i = 0,1,2$. □

C. OBFUSCATING COMPILATION

The compiler works with a database

$$D : \mathrm{DB} = \mathrm{Loc} \to \mathrm{Int}$$

of (32-bit) integer offsets indexed per register or memory location. As the compiler works through the source code, the offset represents by how much the runtime data underneath the encryption is to vary from nominal at that point in the program. All that need be done is to show that what is in D can be controlled as required, and then an information-theoretic argument can be made that the distribution of possible offsets is uniformly flat across compilations. The compiler also maintains a database

$$L : \mathrm{Var} \to \mathrm{Loc}$$

of the locations (registers, memory) for the source code variables’ placements in registers or memory. The compiler, producing FxA machine code, has type:

$$\mathbb{C}^L[\_ : \_] : \mathrm{DB} \times \mathrm{source\_code} \to \mathrm{DB} \times \mathrm{machine\_code}$$

As syntactic sugar, a pair in the cross product will be written $D : s$.

In order to simplify here, details of the management of database L are omitted. It is entirely standard in terms of compiler technique.

C.4 **Sequence:** The compiler works left-to-right through a source code sequence $s_1; s_2$:

$$\begin{aligned} \mathbb{C}^L[D_0 :\ s_1; s_2] &= D_2 :\ m_1; m_2 \\ \text{where } D_1 : m_1 &= \mathbb{C}^L[D_0 : s_1] \\ D_2 : m_2 &= \mathbb{C}^L[D_1 : s_2] \end{aligned}$$

The database D_1 that results from compiling the left source code sequent s_1, emitting machine code m_1, is passed in to the subsequent compilation of the right sequent s_2, emitting machine code m_2 that follows on directly from m_1 in the object code file and its memory image when loaded.

C.5 **Assignment:** An opportunity for varying an offset arises at any assignment to a source code variable x. An offset $\Delta_x = D_1 Lx$ for the data in the target register or memory location Lx is generated randomly, replacing the old offset D_0Lx that previously held for the data at that location. The compiler emits code m_1 for the expression e which puts the result in a designated temporary location **t0** with offset $\Delta_e = D_1\mathbf{t0}$. It is transferred from there to the location Lx by a following add instruction (the **zer** location dummy in the add instruction means that field does not contribute):

$$\begin{aligned}\mathbb{C}^L[D_0 : x{=}e] &= D_1 :\ m_1;\text{add } Lx\ \mathbf{t0}\ \mathbf{zer}\ [i]_{\mathscr{E}}\\ &\quad\text{where } i = \Delta_x - \Delta_e\\ D_1 : m_1 &= \mathbb{C}^L_{\mathbf{t0}}[D_0 : e]\end{aligned}$$

The **t0** subscript for the expression compiler tells it to aim at location **t0** for the result of expression e. That is one of the registers reserved for temporary values.

C.6 **Return:** The compiler at a 'return e' from function f selects a final offset $\Delta_{f_{\text{ret}}}$ (functions f are subtyped by offsets $\Delta_{f_{\text{par0}}}$, $\Delta_{f_{\text{par1}}}$, etc. in their formal parameters and $\Delta_{f_{\text{ret}}}$ in their return value) and emits an add instruction with target the standard function return value register **v0** prior to the conventional function trailer (ending with a jump back to the address in the return address register **ra**). The add instruction adjusts to the offset $\Delta_{f_{\text{ret}}}$ from the offset $\Delta_e = D_1\mathbf{t0}$ with which the result from e in **t0** is computed by the code m_1 compiled for e:

$$\begin{aligned}\mathbb{C}^L[D_0 : \text{return } e] = D_1 :\ & m_1;\text{add } \mathbf{v0}\ \mathbf{t0}\ \mathbf{zer}\ [i]_{\mathscr{E}}\\ & \ldots \qquad \text{\# restore stack}\\ & \text{jr } \mathbf{ra} \quad \text{\# jump return}\\ \text{where } i &= \Delta_{f_{\text{ret}}} - \Delta_e\\ D_1 : m_1 &= \mathbb{C}^L_{\mathbf{t0}}[D_0 : e]\end{aligned}$$

The offset accounted for **v0** is updated in D_1 to $D_1\mathbf{v0}{=}\Delta_{f_{\text{ret}}}$.

The remaining source code control constructs are treated like return. For an if statement, for example, final offsets in each branch are adjusted to match at the join.

The offsets Δ_l at each point in the program are *inputs* to the compilation. One should choose them stochastically with flat probability across the whole 32-bit range.

PROOF (FACT 2). Suppose that at the point just before the FxA instruction I in the program, for all locations l the value $x{+}\Delta_x$ with $[x{+}\Delta_x]_{\mathscr{E}}$ in l varies randomly across recompilations with respect to a nominal value x with probability $p(x{+}\Delta_x{=}X){=}1/2^{32}$, and I writes value $[y]_{\mathscr{E}}$ in one particular location l. That y has an additive component k that is generated by the compilation so as to offset y from the nominal functionality $f(x+\Delta_x)$ by an amount Δ_y that is uniformly distributed across the possible range. Then $p(y{=}Y){=}p(f(x{+}\Delta_x){+}\Delta_y{=}Y)$ and the latter probability is $p(y{=}Y) = \sum_{Y'} p(f(x{+}\Delta_x){=}Y' \wedge \Delta_y{=}Y{-}Y')$. The probabilities are independent (because I is only generated once by the compiler and Δ_y is newly introduced for it), so that sum is $p(y{=}Y){=}\sum_{Y'} p(f(x{+}\Delta_x){=}Y')p(\Delta_y{=}Y{-}Y')$. That is

$$p(y{=}Y){=}\frac{1}{2^{32}}\sum_{Y'} p(f(x{+}\mathrm{d}x){=}Y')$$

Since the sum is over all possible Y', the total of the summed probabilities is 1, and $p(y{=}Y){=}1/2^{32}$. The distribution of $x{+}\Delta_x$ in other locations is unchanged. □

A helpful intuition is that Δ_y has maximal entropy, so adding it in in an instruction completely swamps any other information the instruction might have exposed.

D. SEMANTIC SECURITY

Fact 2 provides the probabilistic setting for semantic security as follows. Consider a probabilistic method F that guesses for a particular runtime value beneath the encryption 'the top bit is 1, not 0', with probability $p_{C,T}$ for program C with trace T. By Fact 2, 1 and 0 are equally likely across all possible compilations C, and the probability F is right is $\mathrm{p}(\mathrm{bit}_{C,T} = 1 \text{ and } F(C,T) = 1) + \mathrm{p}(\mathrm{bit}_{C,T} = 0 \text{ and } F(C,T) = 0)$. Splitting the conjunctions, that is $\mathrm{p}(\mathrm{bit}_{C,T} = 1)\mathrm{p}(F(C,T) = 1\,|\,\mathrm{bit}_{C,T} = 1) + \mathrm{p}(\mathrm{bit}_{C,T} = 0)\mathrm{p}(F(C,T) = 0\,|\,\mathrm{bit}_{C,T} = 0)$. But the method F cannot distinguish the compilations it is looking at as the codes and traces are the same to look at, modulo the (encrypted) values in them. The method F applied to C and T has nothing to cause it to give different answers but incidental features of encrypted numbers and its internal spins of a coin. Those are *independent* of if the bit is 1 or 0 beneath the encryption, supposing the encryption is effective. So $\mathrm{p}(F(C,T) = 1\,|\,\mathrm{bit}_{C,T} = 1) = \mathrm{p}(F(C,T) = 1) = p_{C,T}$ and $\mathrm{p}(F(C,T) = 0\,|\,\mathrm{bit}_{C,T} = 0) = \mathrm{p}(F(C,T) = 0) = 1 - p_{C,T}$, and the probability F is right reduces to $0.5p_{C,T} + 0.5(1 - p_{C,T}) = 0.5$. That is no better than chance.

X. CHANGES FOR REBUTTAL

i Emphasise application area in abstract.

ii Since a referee says the English is "poor" whereas in fact it is excellent (the writers are experienced authors and native British English speakers), we deduce that may be a problem. We were as sparing of words as possible, as we have observed that hardware engineers dislike reading. Here we have put words back in to help the non-native reader. We have done this at a few more points in the text too, but this is the only time we will formally comment on it.

iii Emphasise that we are not playing race-the-fastest, for the benefit of referees with experience of nothing else but that, rather introducing a radically new approach. Others may improve it.

iv Anonymity forces the referees to a choice – either stop making complaints about "self-plagiarism", which is impossible in an anonymous work and would mean the referees have disqualified themselves by breaking the double blind protocol, or just accept that an author will have to longwindedly repeat explanations found in previous work rather than neatly reference it. There is just one best way to explain something. In particular, there is just one best pictorial illustration of a concept, whoever drew something like it first, and it is either that, or do something second best, or throw away anonymity with a self-reference. 'Plagiarism' means 'copying' and no copying takes place in repeated original creation from zero. Copyright law makes the distinction, as does science, else Einstein could not twice draw a picture of a photon passing a train.

v Quibble, because hardware security can not be enough on its own, and referees should not think it is. Try a program that takes twice as long to produce an answer numerically twice as large.

vi Unable because of anonymity to reference our own security papers, we reference an appendix instead, emphasise that security of *data* is formally claimed, not control flow, and go on to show it equivalent.

vii Treat basic security questions early, for referees with a hardware engineering orientation who may otherwise not see a problem. We are not able to logically infer referees' misconceptions when they are not plainly expressed as 'I do not know X, please explain it to me', but we have tried our best to second-guess at the root causes in referees' remarks, and treat those here.

viii Give reference and highlight for referees that doing IEEE floating point encrypted is elsewhere seen as computationally impractical but there is no problem here. FP is done in the pipeline just as integer arithmetic is, with the FPU in the role of the ALU.

ix Deal with a referee's invitation to expound on what role simulation plays and how 'accurate' it is (a meaningless question, all said).

x Devote one paragraph to precisely what is new here, as hardware engineers seem to want headlines.

xi Make clear that the reference is to an old design, for a referee who may be a security expert rather than a computer engineer

xii Make clearer how this number may be calculated, so referees should not think it is made up. 0.5×10^9 comes from the KPU speed as against IBM's 1Hz. Another 30× comes from 32-bitness of the KPU versus 1-bit operations in IBM. Another factor 10× comes from the number of gates to build a full 1-bit adder with.

xiii Sum up the interdependent nature of security in one sentence for referees' convenience at end of Introduction, for those that do not draw abstract threads together easily or did not read the text.

xiv Give some more details of how simulation works, to assuage referees' possible doubts that it is done right. The anonymity requirements forbid us referencing our own books on VHDL and articles on VHDL refinement and so on, so no more can be done.

xv Give intuition as to why changes in OpenRISC instruction set are necessary rather than just state them.

xvi Emphasise for referees that floating point as well as integer is working encrypted, for those that did not absorb the footnote about the IEEE test suite.

xvii Draw more attention to the pseudo-random padding and what it is for, in case a hardware referee weights looking at diagrams higher than reading or thinks if it is not shown in a diagram it is not there.

xviii Give more insight into areas where real effort is required, for the benefit of referees who may be insensible to how hard is software.

xix Give insight that feeding the processor with instructions at the rate it really needs is bottlenecked by legacy issues, and that the buffer in the fetch stage is there to help fix that, as well as give the decode stage a broadside view of several instructions at once.

xx Explain for referees how much of a kludge the inherited fetch stage is, so they get insight into how hard fighting a million lines of simulation code is and where real improvements can be easily bought.

xxi Point out even more forcefully via a recent reference that SGX is broken, for referees that may try ‘what is wrong with SGX?’ to dismiss the paper.

xxii Give the hint that completeness is the ‘real deal’ for computing and homomorphic encryption cannot do it, so just forget the hype.

xxiii For completeness, mention SHEs, in case referees think we do not know that they can be ‘good as’ FHEs for specific fixed calculations, while 1,000$\times$ as fast (which is as fast as a 30Hz Pentium).

xxiv Forward reference to next numbered paragraph for referees to explain what might otherwise seem to be an ‘unjustified claim’.

xxv Clarify for referees without a security background what ‘exposed’ means here.

xxvi Augment Table 6 with a benchmark for no encryption at all. Add RISC machine comparisons in rows at end of table, with corresponding additions in text nearby. This is to comfort referees that simulation is saying normal and believable things (ARM’s own benchmarks possibly exaggerate by $\times 2$ to $\times 4$, via optimised compilation as per Table 6’s O0-6 row, putting our numbers spot on).

xxvii Although we intend always to report failures and unknowns as well as successes, for scientific reporting, ‘admission’ of unknowns seems to be taken badly by hardware referees, who perhaps may be unused to pinpointing what is unknown to them. So we have generally been loath to dwell on unknowns, but this is one point where we say ‘we do not know yet’ why deprefixing does not mostly make up the difference between encrypted and unencrypted running, as we would naively predict. Referees, please note that our seeing that is a research *success*. Identifying an unknown is the first step to knowing it (paraphrasing D. Rumsfeld, Feb., 2002).

xxviii Emphasise that the graph is original, for referees who might think that keeping the same format as in the referenced publications, for the purpose of easier comparison, is ‘plagiarism’. The graph is different. It has different numbers. Gnuplot makes all look alike.

xxix Clean up description of FxA for referee. That single example for multiplication is enough for any mathematician. The explicit list given only leaves out those RISC instructions that have no arithmetic content, like jal, rfe, mtspr, etc., that are (vacuously) fine for FxA as is. You will not get a complete list without a NDA. But you could check the list of target instructions for the compiler. It is at the referenced URL.

xxx Focus the conclusion on referees’ (deduced) interests, using as plain language as possible to say that we have solved the problem of computing securely, at speeds that a person can live with, and we would now like the h/w community to take over. Our job is to say what to do and show to a reasonable approximation that a person skilled in the art could make this work well, your job is to do it.