Superscalar Encrypted RISC

Measure of a Secret Computer

Peter T. Breuer
Hecusys LLC
Atlanta, GA

Jonathan P. Bowen
London South Bank University
London, UK

***Abstract*—Modifying the arithmetic embedded in a processor can cause data to remain in encrypted form throughout processing. The theory has been prototyped in a superscalar pipelined general purpose processor that 'works encrypted', a new approach to encrypted computation that is reported here.**

The prototype runs encrypted machine code on encrypted data in registers and memory and on buses. The objective is to protect user data against the operator, and 'Iago' attacks in general, for those computing paradigms that entail trust in data-oriented computation in remote locations, overseen by untrusted operators, or embedded and unattended.

The modified architecture is 32-bit OpenRISC. It admits any block cipher compatible with the physical word size. We are reporting performance from cycle-accurate behavioural simulations running AES-128 (symmetric, keyed; the US Advanced Encryption Standard) and Paillier-72 (asymmetric, additively homomorphic, no key in-processor) encryptions in a 128-bit word, and RC2-64 encryption (symmetric, keyed) in 64 bits.

I. Introduction

IF the arithmetic in a conventional processor is modified appropriately, then, given certain provisos, the processor continues to operate correctly, but all its states are encrypted [2]. Running the appropriate machine code architecture, it is impossible for even the privileged operator to decipher data or to deliberately fake results for user programs running in such a processor, even though he/she has access[1]. The theory opens a path to engineering a processor that runs 'profoundly encrypted' at near conventional processor speeds, because in principle only one piece of stateless logic in the processor, the arithmetic logic unit, need be changed with respect to a standard design. Data and data addresses in memory, registers and on buses, etc., start encrypted and stay encrypted.

That account may seem counter-intuitive to the engineer who knows from experience that a tiny change in a computer program, or a minor bug in a hardware unit, gives rise to catastrophically wrong program results. Changing *all* the arithmetic should be inherently dangerous: in a processor that works encrypted, not even 4 may be added as-is to a memory address to get the address of the next word along. Indeed, the encryption of a given value is not unique, but varies during processing, and that should give rise to unease on its own account, because adding even an encrypted 0 to an encrypted memory address may mutate its encrypted value, while the – entirely conventional – memory has no knowledge of the encryption that must be used to discount the change.

A sequence of cycle-accurate behavioural models of encrypted processors have been designed, built and tested in order to (i) demonstrate that the theory is correct in itself, and (ii) explore the limits of its applicability. With respect to (ii), it was not known before prototyping if any conventional instruction set would be compatible with encrypted running, and now the situation is known, it is certain that not every program can run encrypted – compilers and other programs that arithmetically transform the addresses of program instructions (as distinct from addresses of program data) are theoretically impossible [3], and exploration of just what applications can run encrypted has only begun. So far the largest application suite ported[2] is 22000 lines of C, but it and every other application ported (now approx. fifty) has worked well.

Behavioural models have provided metrics for guidance and feedback throughout development, and measurements are reported in Section V. Performance-driven development has identified inefficiencies and driven the evolution of an architecture that works measurably well. There are natural obstacles – for example, encrypted code is longer, and byte and half-word accesses are implemented arithmetically – that mean that encrypted running *should* be slower than unencrypted, and the question is by how much. The measured numbers help technology adopters decide their options. That means aiming to design for speed, since the security is proved (Appx. A). Prototyping has articulated design principles that cause the hardware to behave securely [5], without which an encrypted processor would be full of bugs and vulnerabilities.

Simulation. The OpenRISC 'Or1ksim' simulator (http://opencores.org/or1k/Or1ksim) has been modified to run our processor models. It is now a cycle-accurate simulator, 800,000 lines of finalised C code having been added over two years real time (25 years SE effort), through a sequence of eight prototype processor models. The source code archive and history is available at http://sf.net/p/or1ksim64kpu.

Instruction set. The current design runs the 32-bit OpenRISC instruction set (see openrisc.org) encrypted (opcodes are not encrypted), leaving no doubt that the design is capable of

[1] As reassurance, some of the theory is sketched in Appx. A.

[2] The IEEE floating point test suite from http://www.jhauser.us/arithmetic/TestFloat.html.

general purpose computation, which might be questioned if the instruction set were less conventional.

Encryption. The processor has been adapted for Rijndael-64 and -128 symmetric ciphers (the latter is the US Advanced Encryption Standard (AES) [7]), as well as RC2-64 [21] and Paillier-72 [28], an additively homomorphic cipher that runs without keys in the processor. In principle any block cipher with a compatible block size is feasible.

Toolchain. Existing GNU 'gcc' v4.9.1 compiler (github.com/openrisc/or1k-gcc) and 'gas' v2.24.51 assembler (github.com/openrisc/or1k-src/gas) ports for the OpenRISC 1.1 architecture have been adapted. The modified code is at sf.net/p/or1k64kpu-gcc and sf.net/p/or1k64kpu-binutils.

Limits. The designs tested have 64-bit and 128-bit physical word sizes, which means 64/128-bit registers, buses, memory accesses and encryption block size, but that is not a limit. Word widths up to 2048 bits are contemplated with current technology, if memory accesses are paralleled to maintain the transfer rates that are tested with the 64/128 designs (nominally 15ns per memory access and 3ns cache access).

Configuration. Tests are centered about a 15-stage pipeline configuration, 10 stages of which are for the modified arithmetic, but between 1 and 20 arithmetic stages have been explored. Simulations run a nominal 1GHz clock. The memory and cache access times in the parenthesis above are arbitrarily adjustable for testing. For Paillier-72 encryption the arithmetic is 72-bit multiplication modulo a 72-bit number, feasible in 7 to 20 stages. But at 2048 bits Paillier arithmetic would seem to need improbably long pipes – nevertheless, the closest contemporary design to ours is HEROIC [33], [34], a stack machine running encrypted with a 'one instruction' machine code (the 'OI') prototyped with 2048-bit words encrypting 16 bits of data each. It does do the 2048-bit Paillier arithmetic in hardware, so it is possible.

Key management. There is no circuit to read keys once they are in the processor (if keys are needed for the encrypted arithmetic, as for AES but not Paillier), where they configure the hardware functions. Keys will be embedded at manufacture, as with Smart Card technologies [22] or introduced via a Diffie-Hellman circuit [6] or equivalent that loads the key in public view without revealing it to even a privileged observer.

Key management is then a business question, because there are no consequences of running with the wrong key: if user A runs with user B's key, 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, 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. The situation is at its worst when one of A and B is the privileged operator, and the other is an ordinary user, but that is precisely what the platform is intended to defend the user against. So the consequences of key mismanagement are already defended.

The organisation of this article is as follows. After reviewing the competition in Sections II&III, the architecture is described in Section IV and performance in Section V.

II. Related work

The only broadly comparable contemporary is HEROIC [33], [34], running 16-bit arithmetic in Paillier-2048 encryption on a stack machine architecture. Its core does an encrypted addition operation in 4000 cycles and 20us on 200MHz programmable hardware, roughly equal to a 25KHz Pentium.

While stack machines are different from conventional von Neumann architectures, there have been hardware implementations [16], [30] as recently as a decade ago in connection with Java bytecode, though apparently no more since then until HEROIC. HEROIC replaces the standard 16-bit addition by multiplication of 2048-bit encrypted numbers modulo a 2048-bit modulus m. The Paillier encryption $\mathcal{E}$ fits with that because it has the 'homomorphic' property that multiplying the encrypted numbers $\mathcal{E}(x) * \mathcal{E}(y) \bmod m$ is the same as adding the unencrypted numbers $x + y \bmod 2^{16}$:

$$\mathcal{E}(x) * \mathcal{E}(y) \bmod m = \mathcal{E}(x + y \bmod 2^{16}) \tag{1}$$

In generalised form, for some equivalence relation '$\equiv$', as

$$\mathcal{E}(x) \ \mathrm{op}' \ \mathcal{E}(y) \equiv \mathcal{E}(x \ \mathrm{op} \ y) \tag{2}$$

that is the property required in [2] of a modified arithmetic operation op' for correct working of an encrypted processor, so the theory developed in [2] covers HEROIC too.

Both arithmetic and encryption may be varied when (2) governs the design, which is why our architecture, which follows (2), may work with very different block encryptions ranging from AES to Paillier. And while the HEROIC encryption $\mathcal{E}$ as per (1) is deterministic (one-valued), (2) admits non-deterministic (many-valued) encryptions $\mathcal{E}$, which is best practice for encryption, and that is implemented in our design.

There are intrinsic complications with Paillier and other homomorphic encryptions, however. A separate lookup table is required to detect signed overflow, and while that is very feasible for the HEROIC 16-bit arithmetic and deterministic encryption, it is less so for our 32-bit arithmetic and non-deterministic encryption, requiring design tradeoffs to make it possible. Moreover, while HEROIC does encrypted subtraction with the aid of a second lookup table, it is done dynamically in our design, trading the table size burden for slower subtraction.

Encrypted multiplication (and other operations) must be implemented in (encrypted) software under Paillier. The selling point of Paillier, however, is that (1) means that the modified arithmetic in the processor *needs no keys*. There is nothing to hide, and nothing to be seen even via a physical probe. Customers will trade-off processor speed for that.

III. Other work

Intel. Intel's SGX™ ('Software Guard eXtensions') processor technology [1] 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 different memory 'enclaves'. While the

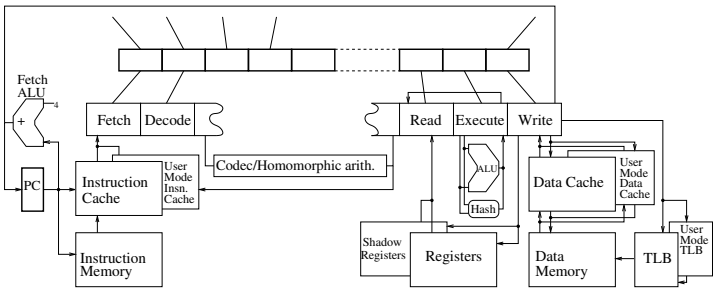

Fig. 1. Pipeline integration with functional units.

enclaves can be encrypted because there are *codec*s (encryption/decryption units) available on the memory path, that is encrypted and partitioned storage rather than encrypted computing, a venerable idea [17], [18].

Nevertheless, SGX machines are often used [31] by cloud service providers where the assurance of safety is a selling point. But the assurance is founded in the customer's trust in electronics designers 'getting it right' rather than mathematical analysis and proof, as available for our and HEROIC's technologies. There are subtle ways for engineering to let secrets slip, via timing changes and power usage [37], for example.

IBM. IBM's efforts at making practical encrypted computation using very long integer lattice-based fully (i.e., additively *and also* multiplicatively) homomorphic ciphers based on Gentry's 2009 discovery [9] also deserve mention. Such ciphers $\mathcal{E}$ extend the equation (1) true for Paillier to cover multiplication as well as addition. 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 [10] on customised vector mainframes with a million-bit word size, about the speed of a 0.03Hz Pentium, but it may be that newer fully homomorphic ciphers based on matrix addition and multiplication [11] will turn out to be more amenable. The product is not going to be capable of arbitrary general purpose computation in any case, just certain finite calculations. An obstacle to computational completeness is that which HEROIC overcomes with Paillier: an encrypted comparison operation is needed, as well as the encrypted addition (and multiplication). HEROIC solves the problem via a lookup table, but that is not feasible for a million-bit encryption.

Cloud. Prototype processors specifically aimed at forms of 'encrypted computation' in the Cloud (e.g., Ascend [8]) do exist, but their internal computation beyond the I/O pins is not encrypted but obfuscated by various physical means, partly as described below (Ascend runs approx. 13.5 times slower in encrypted mode than unencrypted).

Moat electronics. Classically, information may leak indirectly via processing time and power consumption, and 'moat technology' [20] to mask those channels has been developed for conventional processors. The protections may be applied here (and to HEROIC) 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, and statistics are not available to the other mode, so side-channel attacks based on cache-hits [36], [37], are not available.

Oblivious RAM. At the component level, 'oblivious RAM' [24], [26], [27] and its recent evolutions [23], [25]) is often cited as a defense against dynamic memory snooping. That is in contrast to static snooping, so-called 'cold boot' attacks [12], [14], [32] – 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 contents end up encrypted; the address distribution is also uncorrelated in our case. An oblivious RAM remaps the logical to physical address relation dynamically, taking care of aliasing, so access patterns are statistically impossible to spot. It also masks the programmed accesses in a sea of independently generated random accesses. However, it is no defense against an attacker with a debugger, who does not care where the data is stored, and therefore provides no defense against the operator and operating system, which our technology can be proved to do. However, some 'oblivious' behaviour is already in our design, because data addresses are (nondeterministically) encrypted and will vary (indeed, the logical to physical translation may be deliberately remapped at every write to an address). Compiling correctly in part means taking account of that [3], [4].

IV. Architecture

Modes. In user mode, the processor runs on encrypted data and executes the 32-bit part of the OpenRISC 32/64-bit instruction set. In supervisor mode it runs unencrypted and may execute all instructions. Here '64-bit' refers to the arithmetic; instructions are 32 bits long. A 64-bit instruction raises an 'illegal instruction' exception in user mode. User mode has access to 32 general purpose registers (GPRs), and some special purpose registers (SPRs). Attempts to write 'out of bounds' SPRs are silently ignored in user mode, and zero is read. User mode (encrypted) coverage of OpenRISC 32-bit integer and floating point instructions is complete.

In supervisor mode access to available registers is unrestricted. There is no division of memory into 'supervisor' and 'user' parts, so a supervisor mode process can read user data in memory, but the user data will be in encrypted form. The same holds with respect to registers.

Prefix. A *prefix* instruction has been added to the instruction set to carry encrypted immediate data that would otherwise not fit in a 32-bit instruction. Two prefixes are needed for encryptions with 64-bit (and 72-bit) block size, and four prefixes for encryptions with a 128-bit block size, such as AES-128. Compiler strategy should differ with encryption to deprecate storing data in the instruction in favour of reading it from memory, and that it does not makes comparison between encrypted and unencrypted running difficult.

Pipeline. The instruction pipeline in (unencrypted) supervisor mode is the standard short 5-stage fetch, decode, read, execute, write pipeline expected of a RISC processor [29]. In

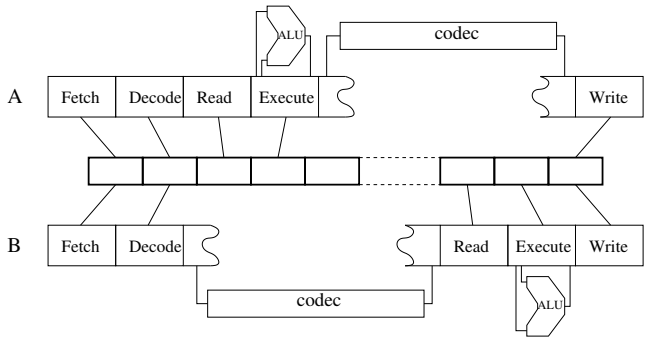

Fig. 2. The pipeline is configured in two different ways, 'A' and 'B', for two different kinds of user mode instructions during encrypted working.

(encrypted) user mode that is embedded in a longer pipeline containing the encrypted arithmetic stages.

The pipeline is configured in two ways, 'A' and 'B', for encrypted running as shown in Fig. 2 (stage hardware is doubled where required). The reason is that, for AES and other symmetric encryptions, a multi-stage codec (configured by an encryption key) is required for the arithmetic. In order to reduce the frequency with which the codec is used, the arithmetic logic unit (ALU) operation is *extended in the time dimension*, so that 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. Longer series mean less frequent codec use. The 'A' configuration is for when codec use must follow arithmetic, the 'B' configuration for codec before arithmetic.

The 'A' configuration is used for store instructions (put an encrypted result into memory) and load instructions (decrypt incoming data from memory). The 'B' configuration is used for instructions with immediate data, which must be decrypted before use. Load and store do not contain immediate data in this variety of OpenRISC, the displacement value from the base address is always zero. Instructions that do not need the codec at all pass through in 'A' configuration, because the early execution is advantageous for pipeline forwarding, avoiding stalls. For AES, the codec covers 10 stages, meaning 10 clock cycles per encryption/decryption.

Shadow. In support, the ALU has a private set of user-mode-only registers that 'shadow' the GPRs (and the few SPRs accessible in user mode) (Fig. 1). These cache the decrypted version of the encrypted data in the 'real' GPRs and SPRs, enabling arithmetic to be carried out unencrypted. The shadow registers are aliased-in for user mode instructions, and aliased out for supervisor mode instructions. Changing the encryption key (if there is one) empties the shadow registers. Otherwise there is no harm in changing from user A to user B without emptying the shadow registers across an interrupt, as argued in Section I: on output the data is in user A's encryption, which user B cannot read. The protocol is proved in [5] to prevent supervisor mode accessing data unencrypted that originated in user mode, and vice versa.

Some supervisor-mode only SPRs have shadow registers. On interrupt GPRs may be copied to these by the supervisor-mode handler and copied back on handler exit, resulting in user-mode context being saved and restored invisibly. User-mode status flags in the processor status register are treated similarly, so supervisor mode never sees user-mode flags.

User data cache. A small user-mode-only data cache retains the unencrypted version of any encrypted data that is written to memory during user mode operation. On load from memory, the cache is checked first. The cache is physically within the processor boundary, so is covered by the processor chip protections from spying or interference (e.g., Smart Card-like fabrication [22], and 'moat' electronics [20]).

User instruction cache. Instructions treated in 'B' configuration have had their immediate data decrypted (for AES and symmetric encryptions in general). The decrypted instructions are cached in a user-mode-only instruction cache, so on a second encounter no decryption is required. The same trick is worked in [15], except cache is shared with supervisor mode there. The caches are flushed on key change.

Address convention. Program addresses are unencrypted (it is data addresses that are encrypted), which potentially is a source of confusion in design and at runtime. A convention handles the issue: unencrypted 32-bit addresses zero-filled to full length are the 'encrypted' form, and they are 'decrypted' to an 'unencrypted' form consisting of the same data with the top bits rewritten to 0x7fff . . . . An instruction such as jump-and-link (JAL) in user mode, which fills the return address (RA) register with the program address of the next instruction, writes the zero-filled address to the real RA register, and the 7fff form to the shadow RA register. Padding or blinding in the real encryption is configured to avoid collisions with both.

TLB. A 'translation look-aside buffer' (TLB) organised by pages is not appropriate in user mode, because encrypted addresses do not cluster, so the user mode TLB (unavailable to supervisor mode) is organised with unit granularity, which means 128 extra bits of location data for each encrypted word. Further, all encrypted addresses are first remapped internally by the TLB to a pre-set range with the allocation ordered by 'first-come, first-served'. Since data that will be accessed together tends also to be addressed together for the first time, this allows cache readahead to be effective.

Addressing hacks. It has turned out to be possible for AES and other symmetric ciphers to pass the unencrypted data address to the memory unit for load and store instructions, with no additional processing. We are nervous of the security implications, so we do not suggest that that should be done. However, the bare 32-bit address can be hashed or encrypted differently, and hashing is being experimented with. The advantage is that global data can then easily be loaded into memory from file by a program loader running in supervisor mode using the hash as address. It is stored in-file with the encrypted data. If the encrypted address were kept instead, the loader would have to run partly in user mode, as a program 'prequel', and it is not clear how that could work. The problem is avoided by not allocating any global ('heap') data in high-

level program source, allocating it on the stack instead. That solution is currently preferred.

V. Performance

The original Or1ksim OpenRISC test suite codes (written mostly in assembler) have established solid benchmarks for encrypted running across years of development now. Most modern performance suites are practically infeasible to compile because they rely on external library support such as linear programming packages and maths floating point libraries, as well as standbys such as 'printf' that must be written and debugged. If those could be ported to compilable code in good time, debugging would take months more (the original OpenRISC gcc compiler has its known bugs, such as sometimes not doing switch statements right, sometimes not initialising arrays right, etc.). Some standard but less evolved benchmarks are running, such as Dhrystone 2.1.

Table I details performance in the instruction set add test of the suite, with RC2 64-bit symmetric encryption, repeating the 2016 test in [5] for comparison. The 64:16:20 mix for arithmetic:load/store:control instructions in user mode (no-ops and prefixes discarded) is approximately the 60:28:12 in the standard textbook [19], so the results are not atypical.

At the time of the earlier test, the program spent 54.8% of the time in user mode, and 52.7% now, which is a 4% (i.e., 2.1/54.8) speed-up in the encrypted running. At the nominal 1GHz clock, pipeline occupation is now $1-20.7/52.7 = 60.7\%$, for 607Kips (instructions per second). That counts no-ops and prefixes too, which are not functional.

The same test with Paillier-72 on the 128-bit architecture shows much worse performance (Paillier does some arithmetic in software, hence the column 2 differences here):

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

The difference is due to more pipeline stalls, not the longer word: running RC2-64 on the 128-bit model gives near the same figures. Paillier arithmetic takes the length of the pipeline to complete, stalling following instructions that need the result as much as 11 stages behind. The disparity is more marked on multiplication, which Paillier does in software:

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

Performance with symmetric encryptions is very sensitive to data-forwarding in the pipeline. This table shows that 33% of processor speed is due to forwarding, while on-the-fly instruction reordering gives only another 3%:

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

In contrast, Paillier shows little sensitivity to forwarding: expected because an arithmetic result is not available before the penultimate stage. The only way to speed up Paillier appears to be to compile multithread programs, so there may be instructions behind that can overtake a stalled instruction.

TABLE I
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%)

Since the 2016 account (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 proceed and pick up the data via forwarding later; (b) the fetch stage has been doubled to get two per cycle and catenate prefixes to the instruction instead of taking pipeline slots; (c) a second pipeline has been introduced to speculatively execute both sides of a branch.

'Flexible staging' (a) takes the cycle count down from 296368 to 259349 cycles on its own. Innovations (b) and (c) then contribute as follows:

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

Branching both ways is not very effective in this test because only 3717 branches were predicted wrongly.

Those tables provide baselines for the AES-128 encryption too via the following Dhrystone 2.1 benchmark equivalences:

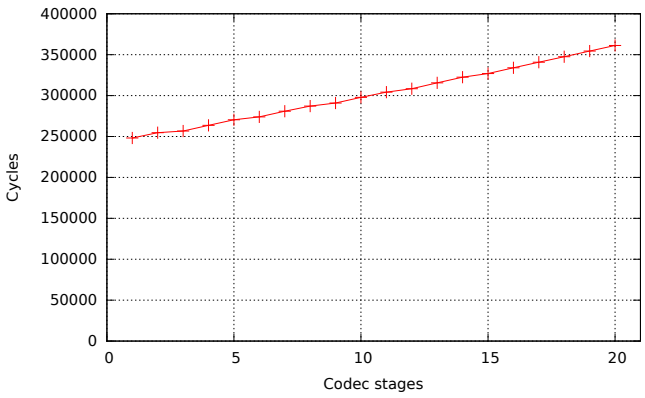

Fig. 3. Number of executed cycles with symmetric encryption against number of stages (cycles) taken up by the codec. Table I is with 10 stages.

Dhrystone v2.1	RC2 (64-bit)	AES (128-bit)
Dhrystones per second	246913	183486
VAX MIPS rating	140	104

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

By those measures, the AES-128 prototype is running as a 320MHz classic Pentium, or 250MHz Pentium M. According to the table at http://www.roylongbottom.org.uk/dhrystone%20results.htm, a 1GHz Pentium M does 523 MIPS and a 100MHz classic Pentium does 32.2 MIPS.

However, the results are very compiler-sensitive, as shown by the variation through optimisation levels O0-O6 for the Pentium M table above, and our compiler is rudimentary. The slowdown for 128-bit AES over 64-bit RC2 is due to the 4, not 2, prefixes for an immediate constant in an instruction.

Results may be extrapolated for longer codecs/more complex encrypted arithmetic in the pipeline. Fig. 3 shows each extra pipeline stage costs approximately 2.5% more cycles.

VI. Conclusion

A superscalar pipelined design prototype for a 32-bit profoundly encrypted processor RISC has been described, embedding RC2 64-bit encryption, the 10-round (Rijndael) AES 128-bit encryption, and Paillier 72-bit additively homomorphic encryption. Registers, memory and buses contain encrypted data in this architecture, which runs an encrypted version of the OpenRISC instruction set. The operator has unfettered and privileged access to internals, yet provably cannot decipher or meaningfully change user data. The prototype, clocked at 1GHz, currently performs at about the level of a 300MHz classic Pentium, in conjunction with AES-128 encryption.

References

[1] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. 2nd Int. Workshop Hardw. Arch. Supp. Sec. & Privacy*, 2013.

[2] 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)*, ser. Lect. Notes Comp. Sci., no. 7781. Heidelberg/Berlin: Springer, Feb. 2013, pp. 123–138.

[3] ——, "Avoiding Hardware Aliasing: Verifying RISC Machine and Assembly Code for Encrypted Computing," in *Proc. 2nd IEEE Workshop Rel. & Sec. Data Anal. (RSDA'14), IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW) 2014*. Los Alamitos, CA: IEEE Comp. Soc., Nov. 2014, pp. 365–370.

[4] ——, "Certifying Machine Code Safe from Hardware Aliasing: RISC is not necessarily risky," in *Proc. Softw. Eng. & Formal Methods (SEFM'13), Colloc. Workshops*, ser. Lect. Notes Comp. Sci., S. Counsell and M. Núñez, Eds., no. 8368. Springer, 2014, pp. 371–388.

[5] P. T. Breuer, J. P. Bowen, E. Palomar, and Z. Liu, "A Practical Encrypted Microprocessor," in *Proc. 13th Int. Conf. Sec. & Cryptog. (SECRYPT'16)*, C. Callegari, M. van Sinderen, P. Sarigiannidis, P. Samarati, E. Cabello, P. Lorenz, and M. S. Obaidat, Eds., vol. 4. Portugal: SCITEPRESS, Jul. 2016, pp. 239–250.

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

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

[8] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proc. 7th ACM Workshop Scalable Trusted Comput. (STC'12)*. New York, NY: ACM, 2012, pp. 3–8.

[9] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theor. Comput. (STOC'09)*, New York, NY, 2009, pp. 169–178.

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

[11] 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)*, ser. Lect. Notes Comp. Sci., R. Canetti and J. A. Garay, Eds. Springer, Aug. 18-22 2013, no. 8042, pp. 75–92.

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

[13] S. Hada, "Zero-knowledge and code obfuscation," in *Proc. 6th Int. Conf. Theor. Applicat. Cryptol. Inform. Sec. (ASIACRYPT'00)*, ser. Lect. Notes Comp. Sci., T. Okamoto, Ed. Heidelberg/Berlin: Springer, 2000, no. 1976, pp. 443–457.

[14] 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.

[15] B. Hampson, "Digital computer system for executing encrypted programs," Jul. 11 1989, US Patent 4,847,902.

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

[17] R. Hartman, "System for seamless processing of encrypted and non-encrypted data and instructions," Jun. 29 1993, US Patent 5,224,166.

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

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

[20] 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.

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

[22] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Proc. USENIX Workshop Smartcard Tech.* Berkeley, CA: USENIX Assoc., May 1999, pp. 9–20.

[23] 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. Program. Lang. & Operating Syst. (ASPLOS'15)*, 2015.

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

[25] 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, NY: ACM, 2013, pp. 311–324.
[26] R. Ostrovsky and O. Goldreich, "Comprehensive software protection system," Jun. 16 1992, US Patent 5,123,045.
[27] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *Proc. 22nd Annu. ACM symp. Theor. Comput.*, ACM. ACM, 1990, pp. 514–523.
[28] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Theor. Applic. Cryptog. Tech. (EUROCRYPT'99)*, ser. Lect. Notes Comp. Sci., no. 1592. Springer, Heidelberg/Berlin, Apr. 1999, pp. 223–238.
[29] D. A. Patterson, "Reduced instruction set computers," *Commun. ACM*, vol. 28, no. 1, pp. 8–21, Jan. 1985.
[30] 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. Springer, Aug. 2004, pp. 917–921.
[31] 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*, May 2015, pp. 38–54.
[32] 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, NY: ACM, 2011, pp. 73–82.
[33] 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.
[34] N. Tsoutsos and M. Maniatakos, "Investigating the application of one instruction set computing for encrypted data computation," in *Proc. Int. Conf. Sec., Privacy & Appl. Cryptog. Eng.* Springer, 2013, pp. 21–37.
[35] M. van Dijk and A. Juels, "On the impossibility of cryptography alone for privacy-preserving cloud computing." *HotSec*, vol. 10, pp. 1–8, 2010.
[36] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proc. 2nd Annu. Comp. Sec. Applic. Conf. (ACSAC'06)*. IEEE, 2006, pp. 473–482.
[37] 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: ACM, 2012, pp. 305–316.

APPENDIX (for publication as requested)

For the reassurance of readers, some of the theory claimed in Section I is sketched here: consider a program C that has been written using only instructions for *addition of a constant* $y \leftarrow x+k$ and branches based on *comparison with a constant* $x<K$. Those are an abstraction of the single instruction $x_1 \leftarrow x_1+k_1$ if $x_2<K_2$ ... from HEROIC, the 'one instruction computer'. Amazingly, despite being able to read (and rewrite) every program, read and write every register and memory location:

Fact 1. *No method of observation exists by which the privileged operator may decrypt the output y of the program C.*

NO METHOD OF OBSERVATION EXISTS ...

Suppose that the privileged operator does have some method of working out what the output y of the program C is, although it is encrypted, having also observed the trace T.

But one can construct a program C' that looks the same to the operator, and gives a runtime trace T' that also looks the same, but C' produces a different output, say $y+7$. The operator's method must predict the same output for C' as for C, so the method is wrong and cannot exist.

The idea is that the operator cannot read encrypted constants in programs C and C', nor encrypted data in traces T and T', so the constants in C may be altered for C' without the operator knowing they are different. Doing that and preserving the trace achieves the objective.

The program C' is easy to construct, imagining that *every* number has had '7' added to it under the encryption.

The encrypted additions $y \leftarrow x+k$ of C still work correctly in C', because they add k to a number that is 7 more than it used to be to get a number that is 7 more than it used to be. They do not need changing.

The comparisons $x<K$ in C do need changing for C' because the new numbers, which are 7 more than they used to be, need to be compared with K' equal to $K+7$ for branches to go the same way as before (care is taken that there are no collisions between the encrypted values k and K by padding or blinding appropriately, so the operator cannot tell either by means of a new collision or lack of an old one that the K have changed).

The argument in the box is independent of the chosen value '7' by which the program output differs from the operator's prediction. It could be any number, with equal probability (see Fact 5 below) as far as the operator is concerned. So the operator's putative method may not even be statistically-based.

An elaboration of the same argument establishes the claim for plural y: the data in any location (register or memory) at any point in the program may be varied by any desired amount from the 'nominal' value, by judicious changes in the encrypted constants in the program C, while maintaining the same runtime trace T (up to the values of encrypted data, which cannot be read by the operator).

Surprisingly, that the operator cannot read any program C means that the operator cannot write one either, at least not one that does what the operator intends. The reason is that if that could be managed, then the operator would know what the output of the program is, and that is impossible, by Fact 1. The argument in the box is not predicated on who wrote program C, so it applies.

Fact 2. *There is no method by which the privileged operator can alter program C using the restricted set of instructions to produce intended outputs y.*

If that argument seems solipsistic, consider that the box argument shows that the hypothetically altered program could produce a different value with the same trace if the encrypted constants in it were different than supposed. So the program cannot be something like 'return $x-x$' (which would return 0 for any x). The box shows that no program using the HEROIC instruction set gives a result independent of the (possibly not any) encrypted constants in it, which the operator cannot read or compose.

That theory depends on the minimalist HEROIC machine code instruction set, but can be extended to other instruction sets too. The

TABLE II
AN FXA MACHINE CODE INSTRUCTION SET

fields						*semantics*
add	r_1	r_2	r_3	$[k]_{\mathcal{E}}$	–	add $r_1 \leftarrow r_2 + r_3 + k$
sub	r_1	r_2	r_3	$[k]_{\mathcal{E}}$	–	subtract $r_1 \leftarrow r_2 - r_3 + k$
mul	r_1	r_2	r_3	$[k]_{\mathcal{E}}$	–	multiply $r_1 \leftarrow r_2 * r_3 + k$
div	r_1	r_2	r_3	$[k]_{\mathcal{E}}$	–	divide $r_1 \leftarrow r_2 / r_3 + k$
cmov	r_1	r_2	r_3		–	conditional move $r_1 \leftarrow \text{flag}\,?\,r_2 : r_3$
li	r_1	$[k]_{\mathcal{E}}$			–	load immediate $r_1 \leftarrow k$
sfeq	r_1	r_2	$[k]_{\mathcal{E}}$		–	set flag if $r_1 = r_2 + k$
sfne	r_1	r_2	$[k]_{\mathcal{E}}$		–	set flag if $r_1 \neq r_2 + k$
sflt	r_1	r_2	$[k]_{\mathcal{E}}$		–	set flag if $r_1 < r_2 + k$
sfgt	r_1	r_2	$[k]_{\mathcal{E}}$		–	set flag if $r_1 > r_2 + k$
sfle	r_1	r_2	$[k]_{\mathcal{E}}$		–	set flag if $r_1 \leq r_2 + k$
sfge	r_1	r_2	$[k]_{\mathcal{E}}$		–	set flag if $r_1 \geq r_2 + k$
bf	j				–	skip j instructions if flag set
bnf	j				–	skip j instructions if flag not set
b	j				–	unconditional skip j instructions
...						

Legend: the r are register indexes or memory locations, the k are 32-bit integers, the j are instruction address increments, '$\leftarrow$' is assignment.

best generalisation is to a 'fused anything and add' (FxA) instruction set (Table II), in which every instruction contains an immediate (encrypted) constant that is added to the result. A 'fused multiply and add' (FMA) instruction was introduced for the purpose of improving parallel processing performance by AMD and Intel in 2011/12/13, with their FMA3/4 instruction sets, and FxA instructions follow that pattern. Even the comparison instructions carry an extra encrypted constant, which is added to one comparand.

Fact 3. *There is no method by which the privileged operator can read a program C constructed using FxA machine code instructions, nor deliberately alter it using those instructions to give an intended encrypted output.*

As already remarked, the argument in the box is 'agnostic' with regard to the possible deviation of runtime values from a nominal value, so in principle the data at any point might be arbitrarily different from the hypothesised values. In reality, however, the program having been written by a human being, a bet that one of the encrypted constants in the program is a 1 and another is a zero would win, making the encryption statistically open to attack.

To combat that, an *obfuscating compiler* has been created which introduces into FxA machine code the arbitrary differences from the nominal values at each point in the program that the box argument says are feasible. It is not appropriate to detail the compiler here, but its operation follows the (extended) box argument, introducing new variations at each non-control instruction in the compiled program via its encrypted constant (see Table II). Then:

Fact 4. *The probability across recompilations that any particular runtime value x is in location l at any given point in the program is uniformly $1/2^{32}$.*

(for a 32-bit system under the encryption).

The intuition to follow for that is that the (additive) noise introduced into each instruction by the compiler's manipulation of the encrypted constant already has maximal entropy over 32 bits, being uniformly distributed across the range. Thus it swamps other information and makes the data written at runtime by the instruction appear uniformly and randomly distributed across possible recompilations, via the well-known Shannon inequality: the entropy of a sum of signals (in modulo arithmetic) is no less than that of any contribution.

That establishes *cryptographic obfuscation* [13]. That is, having the machine code in hand does not enable the operator to guess even a single bit of encrypted runtime data with any statistical advantage over instead having in hand a black box implementation of the code[3]. The operator may be able to identify the author by means of signature control graph constructs in the code, for example, but does not know even statistically what the runtime value anywhere is.

Then by an argument of van Dijk et al. in [35], which reduces the privacy of arbitrary computations in the cloud on behalf of multiple remote users to the question of the cryptographic obfuscation of the machine code running on the server:

Fact 5. *An arbitrary FxA machine code compiled by the obfuscating compiler and executed by our platform in the cloud on behalf of a remote user is private.*

The FMA ('fused multiply and add') machine code instruction is part of the OpenRISC instruction set that the prototype runs and the remainder of the FxA set is emulated by following or preceding one of the standard OpenRISC set by an extra *add immediate* instruction. Configuring the processor to raise an illegal instruction fault on any one of the standard OpenRISC instructions that is not immediately preceded or followed by an add immediate instruction implements FxA exactly, but instructions must be fetched two at a time.

As remarked in Section V, 'deprefixing' (b) doubles the pipeline fetch stage, getting two instructions per cycle, and that provides a convenient place for the hardware check. However, instructions are in any case fetched 64 bits (and more) at a time to the instruction cache, and that is an alternative site for the hardware check. The pair may not straddle a cache line boundary.

The FxA instruction set as shown in Table II does not contain bitwise operations (nor bytewise, nor half-wordwise). Those should be compiled to software functions. Nor does Table II contain floating point or double word operations, but compiling for a 32-bit target without hardware floating point support solves the issue.

However, the prototype does have an (encrypted) 32-bit floating point unit (FPU), implemented in the same way as the 32-bit integer ALU is implemented, and does have the corresponding OpenRISC 32-bit FP instructions. So compilation for a target with 32-bit floating point hardware works as well.

[3]The statement of Fact 4. is slightly stronger than cryptographic obfuscation, in that it shows that the operator has no statistical advantage in formulating a guess at the data anywhere, independently of the availability or not of black-box testing as extra information.