Superscalar Encrypted RISC
A Secret Computer in Simulation

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

[1] Hecusys LLC, Atlanta, GA; Peter.T.Breuer@gmail.com
[2] London South Bank University, London, UK

Abstract. It suffices to change the arithmetic embedded in a processor in order to cause data to remain in encrypted form throughout. The theory has been embodied in a prototype design for a superscalar pipelined general purpose processor that 'works encrypted', a new approach to encrypted computation.
The prototype runs encrypted machine code on encrypted data in registers and memory and on buses. The aim is to protect user data against the operator, and so-called 'Iago' attacks in general, for those computing paradigms that entail trust in data-oriented computation in remote locations, overseen by untrusted operators, or embedded unattended.
The architecture is 32-bit OpenRISC, admitting any block cipher compatible with the physical word size chosen for implementation. We are reporting performance from cycle-accurate behavioural simulations of the design 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 a 64-bit word.

1 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 [14]. It can be impossible for even the privileged operator to infer data or to intentionally alter the result of encrypted programs running in such a processor.[Appx. A] The theory opens a path towards engineering a processor that may run 'profoundly encrypted' at near the speed of a conventional processor, 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 should seem counter-intuitive to an engineer who knows that a tiny change in a computer program, or a minor bug in a hardware unit, can give 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, because adding even an encrypted 0 to an encrypted memory address may mutate its encrypted value, and the – here entirely conventional – memory has no knowledge of the encryption that may be employed in order to discount that 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 right, and (ii) explore the limits of applicability. With respect to (ii), a priori it was not known if any conventional instruction set architecture would be compatible with encrypted running, and now it is known, it is still certain 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 [15], and exploration of which applications can run encrypted has only begun. So far the largest application suite ported is 22000 lines of C [3], but it and every other application ported (now approx. fifty) has worked well.

The models have provided measures for guidance and feedback throughout, and some are reported in Section 5. Performance-driven development has enabled the identification of inefficiencies and the derivation of an architecture that works measurably well. There are 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. Numbers help a potential user of the technology decide between speed and security, which means designing for speed from a secure starting point. But prototyping has also encouraged the articulation of design principles that cause the hardware to behave securely independently of the expression [18], 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 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) [3]), 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.

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

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 in the processor (if keys are needed for the encrypted arithmetic, which is the case for AES), where they configure hardware function. Keys will be embedded at manufacture, as with Smart Card technologies [22] or introduced via a Diffie-Hellman circuit [2] 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 2&3, the architecture is described in Section 4 and performance in Section 5.

2 Related work

The only comparable contemporary is HEROIC [33, 34], running 16-bit arithmetic in Paillier-2048 encryption on a stack machine architecture. While stack

machines are different from conventional von Neumann architectures, there have been hardware prototypes [11, 30] as recently as a decade ago in connection with Java bytecode, though apparently none in the interval. 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, as

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

for some equivalence relation '$\equiv$', that is the property required in [14] of a modified arithmetic operation op$'$ for correct working of an encrypted processor, so the theory of [14] also covers HEROIC. Both arithmetic and encryption may be varied when (2) governs the design, which is why our architecture may work with very different block encryptions, ranging from AES to Paillier. Also, while the HEROIC encryption $\mathcal{E}$ as per (1) is deterministic (one-valued), (2) admits nondeterministic (many-valued) encryptions $\mathcal{E}$, which is best practice for encryption, and followed in our design.

There are intrinsic complications with Paillier, however. A 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 nondeterministic encryption, requiring design tradeoffs to make it possible. And while HEROIC does encrypted subtraction with a second lookup table, it is done dynamically in our design, exchanging the table size burden for slower subtraction.

Encrypted multiplication (and other operations) must be replaced by (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 by a physical probe. Customers will trade-off processor speed for that.

3 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 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 [12, 13].

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

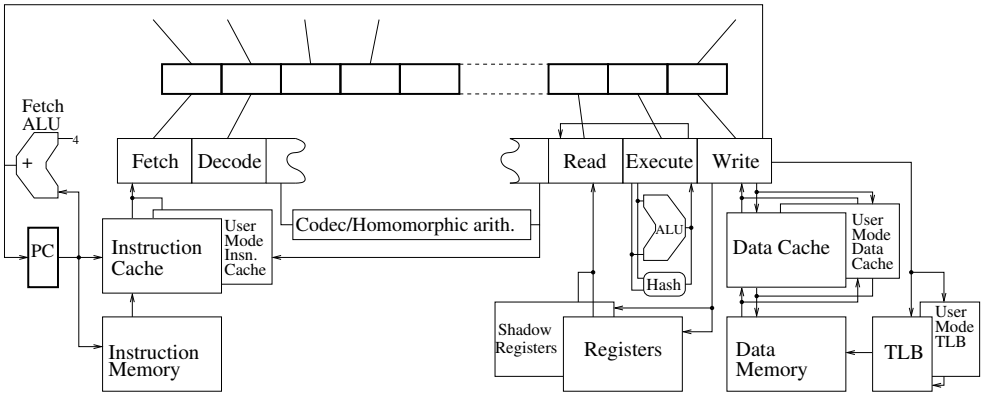

Fig. 1. Pipeline integration with functional units.

in the customer's trust in electronics designers 'getting it right' rather than mathematical analysis and proof, as with our and HEROIC's technologies. There are subtle ways for engineering to give away the secret of an encryption.

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 [5] 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 [6] on customised vector mainframes with a million-bit word size, but it may be that newer fully homomorphic ciphers based on matrix addition and multiplication [7] will eventually turn out to be more amenable. This is not going to be capable of full-blown general purpose computation in any case, just certain finite calculations.

Cloud. Processors aimed at 'encrypted computation' (meaning homomorphically encrypted calculation, usually) in the Cloud (e.g., Ascend [4]) exist too, but their computation is not encrypted but obfuscated, partly as described below.

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 [35, 36], 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' at-

tacks [8,9,32] – 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 [15, 16].

4 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 (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

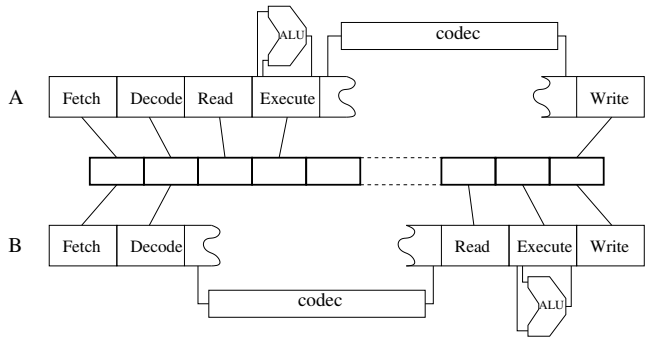

Fig. 2. The pipeline is configured in two different ways, 'A' and 'B', for two different kinds of user mode instructions during encrypted working.

with which the codec is used, 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, so they are unavailable to supervisor mode. 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 1: on output the data is always in user A's encryption, which user B cannot read. The protocol is proved in [18] 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 [10], 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. 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. The padding or blinding associated with encryption avoids both forms.

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 in 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.

5 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 1 details performance in the instruction set add test of the suite, with RC2 64-bit symmetric encryption, repeating the 2016 test in [18] 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		forwarding	
RC2 (64-bit) cycles		✓	×
reordering	✓	296368	412062
	×	315640	441550

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%)

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 instuction.

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:

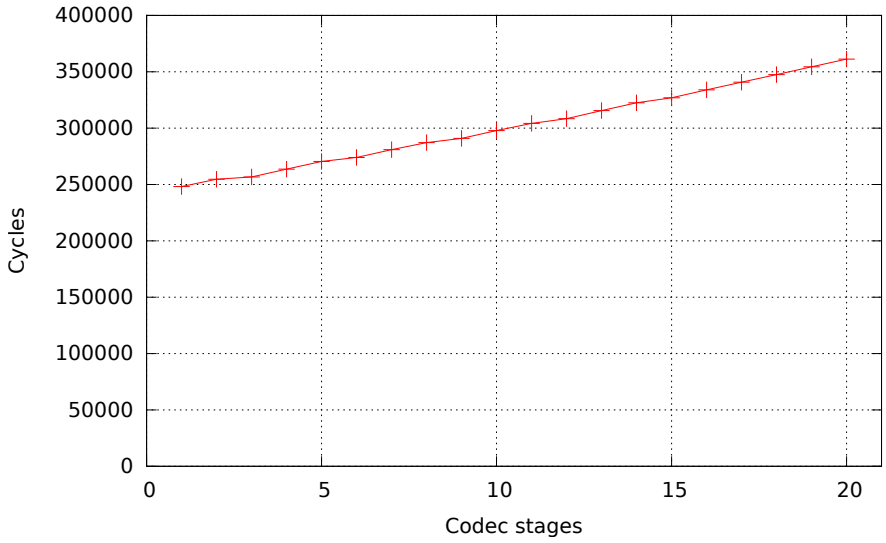

Fig. 3. Number of executed cycles with symmetric encryption against number of stages (cycles) taken up by the codec. Table 1 is with 10 stages.

add test		deprefixing (b)	
RC2 (64-bit) cycles		✓	×
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:

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

According to the table at `http://www.roylongbottom.org.uk/dhrystone%20results.htm`, a Pentium M at 1GHz does 523 MIPS. But the results are compiler-sensitive, as shown by optimisation level O0-O6 for Pentium M, and our compiler is rudimentary. The slowdown for 128-bit AES over 64-bit RC2 is due to 4 prefixes per immediate constant instead of 2. Immediates ought to be deprecated by the compiler.

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.

6 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.

References

1. I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proc. 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. 2013.
2. M. Buer. CMOS-based stateless hardware security module, Apr. 6 2006. US Pat. App. 11/159,669.
3. J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer Verlag, 2002.
4. C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proc. 7th ACM Workshop on Scalable Trusted Computing (STC '12)*, pages 3–8, New York, NY, 2012. ACM.
5. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. 41st Annual ACM Symposium on Theory of Computing (STOC'09)*, pages 169–178, New York, NY, 2009.
6. C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Proc. 30th Ann. Intl. Conf. on Theory and Applications of Cryptographic Techniques (EUROCRYPT'11)*, number 6632 in Lecture Notes in Computer Science, pages 129–148. Springer, Heidelberg, 2011.
7. C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – Proc. 33rd Ann. Cryptology Conf. (CRYPTO '13)*, number 8042 in Lecture Notes in Computer Science, pages 75–92. Springer, Heidelberg, Aug. 18-22 2013.
8. M. Gruhn and T. Müller. On the practicability of cold boot attacks. In *Proc. 8th International Conference on Availability, Reliability and Security (ARES'13)*, pages 390–397. IEEE, Sept. 2013.
9. 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*, 52(5):91–98, 2009.
10. B. Hampson. Digital computer system for executing encrypted programs, July 11 1989. US Patent 4,847,902.
11. D. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the JavaTM virtual machine. In *Proc. 4th IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '01)*, pages 53–59, Washington, DC, USA, 2001. IEEE Computer Society.

12. R. Hartman. System for seamless processing of encrypted and non-encrypted data and instructions, June 29 1993. US Patent 5,224,166.
13. M. Hashimoto, K. Teramoto, T. Saito, K. Shirakawa, and K. Fujimoto. Tamper resistant microprocessor, 2001. US Patent 2001/0018736.
14. P. T. Breuer and J. P. Bowen. A Fully Homomorphic Crypto-Processor Design: Correctness of a Secret Computer. In *Proc. Intl. Symp. on Engineering Secure Software and Systems (ESSoS 2013)*, number 7781 in Lecture Notes in Computer Science, pages 123–138, Heidelberg, Feb. 2013. Springer.
15. P. T. Breuer and J. P. Bowen. Avoiding Hardware Aliasing: Verifying RISC Machine and Assembly Code for Encrypted Computing. In *Proc. 2nd IEEE Workshop on Reliability and Security Data Analysis (RSDA 2014), IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) 2014*, pages 365–370. IEEE Computer Society, Los Alamitos, CA, Nov. 2014.
16. P. T. Breuer and J. P. Bowen. Certifying Machine Code Safe from Hardware Aliasing: RISC is not necessarily risky. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods*, number 8368 in Lecture Notes in Computer Science, pages 371–388, Heidelberg, 2014. Springer.
17. P. T. Breuer and J. P. Bowen. Towards a working fully homomorphic crypto-processor: Practice and the secret computer. In J. Jörjens, F. Pressens, and N. Bielova, editors, *Proc. Intl. Symp. on Engineering Secure Software and Systems (ESSoS 2014)*, number 8364 in Lecture Notes in Computer Science, pages 131–140. Springer, Heidelberg, Feb. 2014.
18. P. T. Breuer, J. P. Bowen, E. Palomar, and Z. Liu. A Practical Encrypted Microprocessor. In C. Callegari, M. van Sinderen, P. Sarigiannidis, P. Samarati, E. Cabello, P. Lorenz, and M. S. Obaidat, editors, *Proc. 13th Intl. Conf. on Security and Cryptography (SECRYPT 2016)*, volume 4, pages 239–250, Portugal, July 2016. SCITEPRESS.
19. K. Hwang. *Advanced Computer Architecture*. Computer Science. Tata McGraw-Hill Education, India, 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 S. Vaudenay, editor, *Proc. 5th Intl. Workshop on Fast Software Encryption (FSE '98)*, pages 206–221, Heidelberg, Mar. 1998. Springer.
22. O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proc. USENIX Workshop on Smartcard Technology*, pages 9–20. USENIX Association, Berkeley, CA, May 1999.
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. Intl. Conference on Architectural Support for Programming Languages & Operating Systems (ASPLOS'15)*, 2015.
24. S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Proc. Theory of Cryptography*, pages 377–396. Springer, Heidelberg, 2013.
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. Computer & Communications Security (SIGSAC'13)*, pages 311–324, New York, NY, 2013. ACM.
26. R. Ostrovsky. Efficient computation on oblivious RAMs. In *Proc. 22nd annual ACM symp. on Theory of Computing*, pages 514–523. ACM, ACM, 1990.

27. R. Ostrovsky and O. Goldreich. Comprehensive software protection system, June 16 1992. US Patent 5,123,045.
28. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. Intl. Conf. on Theory and Application of Cryptographic Techniques (EUROCRYPT'99)*, number 1592 in Lecture Notes in Computer Science, pages 223–238. Springer, Heidelberg, Apr. 1999.
29. D. A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, Jan. 1985.
30. M. Schoeberl. Java technology in an FPGA. In J. Becker, M. Platzner, and S. Vernalde, editors, *Proc. 14th Intl. Conf. on Field-Programmable Logic and its Applications (FPL 2004)*, pages 917–921, Heidelberg, Aug. 2004. Springer.
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. Security & Privacy*, pages 38–54, May 2015.
32. P. Simmons. Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In *Proc. 27th Annual Computer Security Applications Conference (ACSAC'11)*, pages 73–82, New York, NY, 2011. ACM.
33. N. Tsoutsos and M. Maniatakos. Investigating the application of one instruction set computing for encrypted data computation. In *Proc. Intl. Conf. on Security, Privacy & Applied Cryptography Eng.*, pages 21–37. Springer, 2013.
34. N. G. Tsoutsos and M. Maniatakos. The HEROIC framework: Encrypted computation without shared keys. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(6):875–888, 2015.
35. Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proc. 2nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 473–482. IEEE, 2006.
36. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proc. ACM Conference on Computer and Communications Security (CCS 2012)*, pages 305–316, New York, NY, 2012. ACM.

A Unreadability and Unwritability of encrypted programs

For the convenience of readers, some theory mentioned in Section 1 is sketched here: consider a program C that has been written using only the machine code instructions for *addition of a constant* $y \leftarrow x+k$ and branches based on *comparison with a constant* $x<K$. Those two, together with control instructions, are sufficient to perform any computation, evidenced by the single combined instruction that does $x_1 \leftarrow x_1+k_1$ if $x_2<K_2$ ... in the 'one instruction computer' of [34] (and also as evidenced by Conway's well known 'Fractran' language, in which it is also the only instruction):

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

Proof. Suppose for contradiction that the privileged operator has some method $f(T, C)$ of knowing what the output y of the program is, although it is encrypted, having observed the trace T. Imagine, however, that every number has '7' added to it under the encryption. The additions $y \leftarrow x+k$ in the program still make sense, adding 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. The comparisons $x<K$ in the program need changing, however, because the new numbers, which are 7 more than they used to be, need to be compared with K' equal to $K+7$ for the program to still make sense. So we modify the branches in the program to compare with K' instead of K. To the privileged operator, the new program code C' 'looks the same,' $C' \sim C$, because one encrypted number is as meaningful as another (we take care that there are no collisions between the encrypted values k and K used in the program code 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 underneath), and the program trace T' is the same up to the encrypted numbers that occur, which the operator cannot read, so it looks the same, $T' \sim T$, and the operator must see the same outcome from the method $f(T', C') = f(T, C)$ and deduce that the output is $f(T', C') = f(T, C) = y$. Yet the output is not y but $y + 7$.

There are many ways to read the output of a program that is written using an *un*restricted set of instructions. For example, if the program ends 'return $x - x$', then the result is readably (an encryption of) zero. But binary operations are out above.

An elaboration of the proof establishes the proposition for plural outputs y too. Now consider program C again. For definiteness suppose that the K and k come from disjoint subspaces of the cipherspace, so do not collide, and both subspaces are disjoint from data circulating in-processor. That can be arranged by incorporating two type bits in the padding under a symmetric encryption or in the blinding factor of a homomorphic encryption.

Proposition 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.*

Proof. Suppose for contradiction that the operator produces a new program $C' = f(C)$ that returns (encrypted) y. Then its constants k are found in C and its constants K likewise, because f has no way of arithmetically combining them (the disjoint subspaces condition means they cannot be combined arithmetically in the processor and the operator does not have the encryption key). Proposition 1 (plural y) says the operator cannot read outputs y of C', yet knows what they are.

These results may be extended to cover arbitrary additions, subtractions, multiplications, left shifts, provided each instruction is followed by addition of some constant k. The idea is that an observer cannot know if the k is 0 or some other value that compensates for a '+7' in the circulating data. The reasoning also applies for arbitrary comparison operations, not just '$< K$'. But knowing 0 and that some value is not 0 permits reading/writing – see [17] for a generic defence via typed arithmetic.

The status of division, remainder, right shift and the bitwise logical operations with respect to the above is currently unknown. However, right shift is distinguished from left shift by the sign of the operand under the encryption, so the two kinds of instruction are not distiguishable to an observer, and one may follow [4] and claim the result 'by virtue of obfuscation'.

If the observer may construct their own program using arbitrary instructions, and run it through the processor, then that constitutes a different avenue of attack. The attacker may, for example, construct an encrypted zero with the program 'return $x - x$', using any observed x as seed. That may possibly be leveraged in an attack on writability, because knowing a zero contravenes the statement of Proposition 1, on which Proposition 2 relies.

But suppose $0/0 = 0$ and $0\%0 = 0$, and unary bitwise inversion and logical negation are deprecated in favour of $\sim x = 0\text{xffffffff}\hat{}x$ and $!x = x?0{:}1$. Then all operations, plus, minus, mutiplication, shift, etc., produce 0 when all inputs are 0.

Proposition 3. *Zero is the only value of encrypted results y that a privileged attacker can produce intentionally and certainly by creating a program D using arbitrary instructions and inputs x extracted from or results from an observed program C built from a restricted set of instructions as decribed above.*

Proof. Proposition 1 (in plural form) shows that the inputs x can be anything at all as far as an observer knows. Set all x to (encrypted) 0, and all independent variables and constants in program D to zero too, which may be the case as far as the attacker knows. Then the output is (encrypted) 0 through every arithmetic operation in program D, and every branch chooses between different values that are all 0. Since 0 is what the outputs can be and the outputs y are produced with certainty, 0 must be what the attacker intended as y.

Proposition 4. *Zero is the only value for encrypted results y that a privileged attacker can read with certainty from a program D built from arbitrary instructions, plus its runtime traces.*

Proof. If the results y could be read, then the attacker could (by copying) write the program D that produced y, contradicting Proposition 3.