The Secret Processor Will Go to the Ball
Benchmark Insider-Proof Encrypted Computing

Peter T. Breuer
Hecusys LLC
Atlanta, GA

Jonathan P. Bowen
London South Bank University
London, UK

Esther Palomar
Birmingham City University
Birmingham, UK

Zhiming Liu
South West University
Chongqing, China

***Abstract*—Appropriately modifying the arithmetic in a processor causes data to remain in encrypted form throughout processing. That principle is the basis for the design reported here, extending our initial reports in 2016. The design aims to prevent insider attacks by the operator against the user. Progress and practical experience with the prototype superscalar pipelined RISC processor and supporting software infrastructure is reported. The privileged, operator mode of the processor runs on unencrypted data and has full access to all registers and memory in the conventional way, facilitating operating system and infrastructure development. The user mode has restricted access rights, as is conventional, but the security barrier that protects it is not based on access but on the fact that it runs through-and-through on encrypted data. It has been formally shown impossible for operator mode to read (or write to order) data originating from or being operated on in the user mode of the processor in this context.**

Cycle-accurate measures based on our stable toolchain for AES-128 encrypted computation on a 128-bit architecture, and RC2-64 on 64-bit, show 104-140 Dhrystone MIPS respectively for a nominal 1 GHz base clock, equivalent to a 433-583 MHz classic Pentium. This paper aims to alert the secure hardware community that encrypted computing is both possibly practical and theoretically plausible.

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 [3], 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. Memory (RAM) is not part of a processor, but since the processor's outputs to as well as its inputs from RAM and other peripherals are encrypted, memory content is encrypted too.

Running an appropriate machine code instruction set, it turns out that it is mathematically impossible for the operator to infer from a user's computation either statistically or logically or even by experiment what the user's encrypted data means, despite having read and write access to the user's data and program code (see Section 6). Those recent theoretical results have put on a firm footing the security of processor designs that depend on a modified arithmetic and encrypted working, where before they were only intuitively safer. It was always probable from an engineering point of view, however, that such a processor would run fast or could be made to with current technology. That is because, in principle, only one piece of stateless logic, the arithmetic logic unit (ALU), needs to be changed from a conventional design – the rest remains the same.[1] This paper provides experimental data from our prototype to support that view.

If the reader is to take away one thing from this paper, it should be the understanding that in supervisor mode[2] this processor runs unencrypted, while in user mode it runs encrypted. Encryption, not access, is the security barrier and it protects the user against the operator. If successful, it protects code running on behalf of a user in user mode on the processor against 'insider' attacks that may reveal what the user data is underneath the encryption from a human or program with access to the operator mode of running. The agent behind an attack may be a subverted operating system, or a bribed system administrator in the computer room. The operator mode is the mode in which the processor starts up in when it is switched on, it is the mode in which it loads its operating system code, and it is the mode it momentarily switches to when servicing hardware interrupts or system calls while a program runs in user mode. It is the mode your processor switched to just now in order to read from disk the bytes that you are reading in your PDF viewer.

The successful operation of any modern computer system depends on a continuous and close cooperation between code running in user and operator modes on the same processor, so 'protecting' the user from the operator is apparently a difficult concept. In operator mode, for example, the processor must alter the mapping of virtual to physical memory addresses and the access rights for each section of memory – it is 'all-powerful' in concept, except for the single line item that in this processor it 'runs unencrypted', with unencrypted inputs, outputs and intermediate states, while user mode 'runs encrypted'. It is not claimed that user mode running cannot be interfered with by the operator – it can. The operator can wipe memory completely, for exam-

1. Other minor changes follow from encrypted working. Section 4 discusses changes to the RAM addressing hardware subsystem in particular.
2. Supervisor/operator are synonyms for 'unrestricted access' mode here.

ple. The formal result is that the operator cannot know what user data is beneath the running encryption (Section 6). That also logically implies it cannot be rewritten deliberately to a value beneath the encryption that is defined independently, such as π, or the key for the encryption (same section).

This approach is simple, and its aim, as well as fast running, is to permit a clear security analysis, which has come out favourably [7], [6] albeit only in the context of an appropriately designed processor instruction set and a compiler that varies machine code in such a way that it eliminates statistical biases (Section 6). Otherwise there would be simple *known plaintext attacks* (KPAs) [2] based on the principle that $x-x=0$, for example, however it is encrypted, or on the statistical fact that human programmers use $0, 1$ more than any other constants in their code.

The medium-term goal is a secure platform for remote computing in the cloud [35], perhaps also for embedded systems such as automobiles or uranium centrifuges. Experience guides the understanding of what is possible for this system. Remote ('batch') computing is an initial target because predetermined codes with a fixed number and depth of iterations are involved – 'encrypted matrix multiplication' is a fashionable example [37]. Batch working allows the encryption to be changed between runs, which would make attacks formally much harder. It is more difficult to change the code or the encryption or its key, securely and reliably, in a continuously running and critical environment such as a driving automobile, but we believe it will be possible. In the batch mode, the remote user compiles and encrypts the program, sends it for remote execution along with encrypted inputs, and receives back encrypted outputs. In this scenario The key is either already in the machine or loaded in public view via secure hardware, but that is not the concern of this paper on computation (see Section 2.10 for a discussion).

The studies here are based on an extensive sequence of behavioural models, beginning in 2009 with a demonstration that dropping a changed ALU into a detailed model in Java of a pipelined processor (http://sf.net/p/jmips/) gave rise to encrypted running (http://sf.net/p/kpu/). The confirming theory was not published until 2013 [3]. From 2014 to 2016, the existing simulator or1ksim (http://opencores.org/or1k/Or1ksim) for the OpenRISC standard processor architecture was modified to 64-bit and now 128-bit operation and cycle-accurate simulation covering a full processor pipeline. The aim has been to (i) demonstrate that the principle of working is correct to engineers who often do not understand or accept mathematical proofs or formally-oriented computer science, and (ii) explore its limits. With respect to (ii), it was unknown beforehand if conventional instruction sets and processor architectures would be compatible, and that may now be taken to be confirmed in great part. It has also become clearer, however, that not every program can run encrypted in this context – compilers and programs that arithmetically transform the addresses of program instructions (as distinct from addresses of program data) must run unencrypted because program addresses are unencrypted. That prevents what would otherwise be a KPA on encrypted but predictable address sequences. The largest application suite[3] ported so far is 22,000 lines of C, but it and every application ported (now about fifty), has worked well, surprising the authors.

Our accurate models have provided good metrics via the burgeoning software infrastructure and the measures are reported here. The standard Dhrystone v2.1 benchmark shows 104-140 MIPS running encrypted, matching a 433-583 MHz classic Pentium. But the paper's particular objective here is to summarise the state of knowledge in a secure hardware engineering forum and convince that it does work, encouraging the community's focus and scrutiny in future.

The organisation of this article is as follows. Section 2 encapsulates the processor design and working in bullet points for the reader. The aim is to address early on what experience tells us are common misconceptions based on analogies with other security devices. There should be no relevant analogies. In particular, encrypted memory ('oblivious RAM', 'ORAM' [27]) has nothing to do with this device – memory is not part of a processor. Nor does Intel's 'SGX'™ range of machines, which use keys to control access to different (encrypted) memory regions but do not run encrypted, have to do with it. There is no relevance for key-management here – why is discussed in 2.10 – and this paper does not discuss it. Security engineers may later design key management as they wish. The closest contemporary related experimental processor architectures are discussed in Section 3. Security engineering considerations in putting the computational principle into workable practice are described in Section 4 and this is the most relatable section for an engineer, but those details are not crucial as they can and should be changed to suit newer technologies. In particular two 'tricks' of implementation are described (first written down in 2016 in [4] and [5]) that are intended to restore good performance in this context to what is intentionally an old-fashioned processor architecture. The intention is that designers will do better than us by applying the principles expressed here to more contemporary architectures. Further hardware optimisations are described in Section 5. Section 6 sets out the modified RISC [29] instruction set that makes encrypted computation secure, in combination with an 'obfuscating' compiler, briefly described.

2. Summary of design and working

This section summarises the processor design and working in 'touchstone points' for the reader to take forward and also refer back to as necessary.

2.1 **Architecture.** The basic layout, described in [4], is the classic single pipelined RISC processor of [29], clocked at a nominal 1 GHz with 3 ns cache. Register layout and functionality follows the OpenRISC v1.1 specification (see openrisc.org), with 32 *general purpose registers* (GPRs) and up to 2^{16} *special purpose registers* (SPRs). Some SPRs with control/monitor functions are modified for security as described in Section 4. Registers and buses are 64 or 128

3. IEEE floating point test suite at http://jhauser.us/arithmetic/TestFloat.html32-bit OpenRISC floating .

bits wide (it differs per model) for encrypted 32-bit or unencrypted 64-bit data.

The prototypes have all incorporated speculative branch execution/prediction, and data forwarding along the pipeline in the same clock (bypassing registers). Successive design iterations have incorporated extra features, such as on-the-fly instruction reordering.

2.2 **Modes.** The processor operates in two modes: *user* and *supervisor* (aka 'operator'), as per the OpenRISC specification. User mode works encrypted on data that is 32-bit beneath the encryption and supervisor mode works unencrypted on 32- or 64-bit data.

2.3 **Adversaries.** The operator is the adversary who tries to read the user's data, and/or rewrite it. The notion of 'operator' is conflated with the supervisor mode of operation of the processor, in which instructions have access to every register and memory location. The idea is that, as the most privileged user, 'operator' stands in for all, in that user data that is secure from the operator is secure from all.

2.4 **Simulation.** The open source OpenRISC 'Or1ksim' simulator, available from http://opencores.org/or1k/Or1ksim, has been modified to run the processor models. It is now a cycle-accurate pipeline simulator, 800,000 lines of C code having been written over 2 years real time and 25 years estimated software engineering effort, through 8 processor prototypes. The source code archive and development history is available at http://sf.net/p/or1ksim64kpu.

2.5 **Instruction set.** In user mode, the processor runs the 32-bit OpenRISC instruction set modified for encrypted operation. Opcodes and register indices are not encrypted, but a prefix instruction has been introduced that allows a following instruction to contain an encrypted constant, which otherwise would not fit in the 32-bit long instruction. In supervisor mode, the (32-bit long) OpenRISC instructions for 64-bit arithmetic on unencrypted data are available.

2.6 **Security of computation.** Adapting all the standard OpenRISC instruction set for encrypted working has confirmed that it is possible to write (unencrypted, supervisor mode) operating system support for user programs (running encrypted). The operating system generally does not need the decryption of a user datum to do what is required (e.g., output it, encrypted, as is). But the experience has clarified that conventional instruction sets are inherently insecure 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. Then any encrypted y may be constructed by repeatedly applying the machine's addition instruction. By comparing the encrypted $1, 2, 4$, etc. obtained with an encrypted z using the instruction set's comparator instructions (testing $2^{31} \leq z$, $2^{30} \leq z$, ... in turn and subtracting whenever it succeeds), the value of z can be efficiently deduced. This is a *chosen instruction attack* (CIA) [31]. Part of the novel contribution of this paper is a 'FxA' instruction set for encrypted RISC against which every attack fails, in that it is no better than guessing (Section 6).

2.7 **Encryption.** The prototypes models have been tested fitted with Rijndael-64 and -128 symmetric encryption (the latter is the US *advanced encryption standard* (AES) [9]), RC2-64 [23] and Paillier-72 [28]. The last is an additively homomorphic[4] cipher that runs without keys in the processor. In principle any 'reasonable' block cipher with a block size that fits in the machine word may be integrated in the pipeline. For symmetric encryptions, multistage en-/decryption hardware is fitted in the pipeline.[5] For homomorphic encryptions a multistage arithmetic unit occupies the same space. All encryptions are one-to-many. For symmetric encryptions, pseudo-random padding under the encryption is generated by hashing operands. For Paillier, 'blinding' multipliers are generated instead.[6]

The choice of encryptions has been dictated by the development path. The open source Or1ksim simulator had to be expanded from 32 bits to 64 (as well as made cycle-accurate and pipelined) and at that point 64-bit ciphers could be handled. The OpenRISC instructions require two 32-bit prefixes per instruction for 64 bits of encrypted data. Two prefixes is also sufficient for 72 bits of encrypted data, so Paillier-72 could be accommodated without further toolchain changes, but it meant doubling processor path widths from 64 to 128 bits to hold 72-bit data. AES-128 then became possible, requiring four 32-bit prefixes per instruction.

Paillier-72 is insecure in practical terms but has served to investigate use of a homomorphic encryption in this setting. Paillier does not become as secure as AES-128 until 2048 bits, but 2048-bit Paillier arithmetic would use too many pipeline stages for practicality. Nevertheless, the closest competing design is HEROIC [34] (see Section 3), a stack machine running encrypted with a 'one instruction' machine code and 2048-bit words encrypting 16 bits of data. It does 2048-bit Paillier arithmetic in hardware, so it is possible (HEROIC runs 4000 cycles of 200 MHz hardware per arithmetic operation).

2.8 **Toolchain.** The 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 OpenRISC v1.1 have been adapted for the encrypted instruction set. Executables are standard ELF format. The source codes are at sf.net/p/or1k64kpu-gcc and sf.net/p/or1k64kpu-binutils. Only the assembler needs to know the encryption key.

2.9 **Limits.** Word width (i.e., encryption block size) up to 2048 bits is contemplated with current technology. Memory paths would need to be appropriately broadened and accesses paralleled.

2.10 **Key management.** There is no means to read keys once they have been embedded in the processor, where they configure the hardware functions. In a design nearer production, keys may be embedded at manufacture, as with Smart Cards [25] or introduced via a Diffie-Hellman circuit

4. 'Homomorphic' in the Paillier encryption means that multiplication of encrypted numbers corresponds to addition of unencrypted numbers.

5. An AES round is budgeted at 1ns in the models. That is 10 pipeline stages occupied by the encryption/decryption hardware. The Intel/AMD 'ASENC' AES round instruction takes 0.95 ns (4 cycles at 4.2 GHz) on Skylake cores (Table C-9 of [21]), so this is realistic.

[8] that securely loads the key in public view.

Note there is no *direct* consequence of running with the wrong key because if user A runs with user B's key, then user A's program will produce rubbish, as the processor arithmetic will be meaningless; if user A runs user B's program while user B's key is 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, and user A can neither supply nor understand that. Security depends not on access but on whether A, who may be the operator, can leverage observations of B's computations to learn about the encryption, and that is answered in Section 6 – negatively, for the right instruction set.

2.11 **Security guarantees.** A hardware protocol described in [5] guarantees that data originating in user mode can never be seen in unencrypted form in supervisor mode, and conversely (see 4.3^{A1}).

3. Related platforms

HEROIC [34] is the most comparable contemporary platform, running a 16-bit machine in Paillier-2048 encryption [28] on a stack-based architecture. Its core does encrypted 16-bit addition in 4000 cycles and 20 μs on the foundation of 200 MHz programmable hardware, equivalent to a 25 KHz Pentium (one 32-bit addition in 40 μs).

Stack machines are different from conventional von Neumann architectures and are not manufactured, but there have been hardware prototypes in connection with Java [17]. HEROIC works by substituting 16-bit addition by multiplication of 2048-bit encrypted numbers modulo a 2048-bit modulus m. Multiplying above the Paillier encryption $\mathscr{E}$ is the same as adding beneath the encryption:

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

A difficulty is that the addition on the right is not mod 2^{16}, so the sum has to be renormalised mod 2^{16} under the encryption, which accounts for half the cycles taken. It is done by subtracting 2^{16} via (+) and looking up a 'table of signs' for encrypted numbers to see if the result is negative or positive. To facilitate that, HEROIC encryption is one-to-one, not one-to-many,[6] or the table would be too large. It is already 16M bytes in size ($2^{16} \times 2048$ bits). The same table is also used for comparison operations (less than, etc).

That technique is also used with Paillier in our models, except that the table of signs is too large to site locally with current technology (at $2^{32} \times 72$ bits times the number of aliases per encryption), so signs are calculated outside the simulation and cached.

Encrypted multiplication and other operations are subroutines under Paillier. The 'selling point' is that (+) means

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

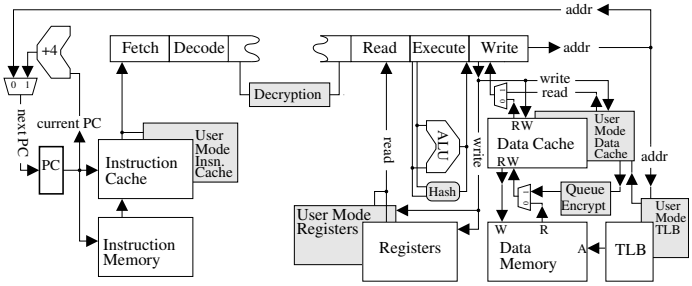

Figure 1. User mode 'shadow' units (shaded).

that the modified arithmetic in the processor *needs no keys*. But despite the headline, the table of signs amounts to a key that must be changed per user.

3.1 **Ascend** [10] protects instructions and data from the operator by both cryptographic and physical means. Code on the way to the processor is encrypted, data I/O is encrypted and the processor runs in 'Fort-Knox'-like isolation, matching pre-defined statistics on observables. Communication with memory is encrypted.

Physical isolation plus encrypted memory has emerged many times (e.g., [19]) and success means doing it as well as Ascend does. Otherwise side-channels such as cache-hit statistics [36] and power drain [24] can leak information. Ascend runs RISC MIPS instructions [30] and slows down by 12-13.5$\times$ in encrypted mode with AES-128 (absolute speeds are not given in [10]), as compared to 10-50% slowdown for our models (Section 5).

3.2 **Intel**'s SGX™ ('Software Guard eXtensions') processor technology [1] is often cited in relation to secure 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 [18], not encrypted computing.

SGX machines are used [32] by cloud service providers where assurance of safety is a marketing point. But that is founded in customer belief in electronics designers 'getting it right' rather than mathematical analysis and proof, as for our and HEROIC's technologies (see Section 6). Engineering may leak secrets via timing variations and power use and SGX has recently fallen victim [15].

3.3 **IBM**'s efforts at making practical encrypted computation using very long integer lattice-based *fully homomorphic encryptions* (FHEs) based on Gentry's 2009 cipher [11] deserve mention. An FHE $\mathscr{E}$ extends the Paillier equation (+) to multiplication on the right. But it is 1-bit, not 16- or 32-bit arithmetic under the encryption. The 1-bit logic operations take of the order of 1s [12] on customised vector mainframes with a million-bit word, about equivalent to a 0.003 Hz Pentium, but it may be that newer FHEs based on matrix addition and multiplication [13] will be faster. The obstacle to computational completeness is that which HEROIC overcomes with its 'table of signs': encrypted comparison with plain 1/0 output is needed, as well as

the encrypted addition (and multiplication), but HEROIC's solution is not feasible for a million-bit encryption.

3.4 **Moat electronics.** Classically, information may leak indirectly via processing time and power consumption, and 'moat technology' [22] 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 [36] are not available.

3.5 **Oblivious RAM** (ORAM) [27] and its evolutions [26] is often cited as a defense against dynamic memory snooping. That is in contrast to static snooping, so-called 'cold boot' attacks [16] – physically freezing the memory to retain the contents when power is removed – against which HEROIC, SGX and our technology defend because memory content is encrypted. Also, in our technology, data addresses are encrypted and vary during running. ORAM extends that by continuously remapping the logical to physical address translation, taking care of aliasing, so access patterns are masked. It also hides programmed accesses among randomly generated accesses. But it is no defense against an attacker with a debugger, who does not care where the data is stored. So it does not defend against the operator and operating system, as the technology here does.

4. Engineering for security

Two major 'tricks' of implementation for good performance were described in [4] in 2016 and are summarised below in 4.1, 4.2.

4.1 **Dual pipeline configuration.** There are two configurations of the pipeline, 'A' and 'B', for encrypted running with symmetric encryption (Fig. 2). There is only space for one (multi-stage) encryption/decryption unit and some instructions need encryption after the execute stage ('A'), some need decryption before ('B'). A variant 'A' configuration is used for Paillier (Fig. 2 top).

4.2 **The arithmetic logic unit** (ALU) operation is *extended in the time dimension* to cover a series of consecutive (encrypted) arithmetic operations in user mode. The first of a series is associated with a decryption event and the last with an encryption event (note that by 'arithmetic operations' is meant the arithmetic stages of individual instructions, not the whole instructions). That reduces the frequency with which the encryption/decryption unit is used.

4.3 **ALU operation** is supported by a hardware protocol described in [5]. Shadow registers/caches (Fig. 1) are managed as follows:

Protocol	*Shadow units are aliased-in for user mode.*	(*)
Invariant	In user mode, each instruction expects and puts encrypted values (or a neutral placeholder) in non-shadow registers and unencrypted values in shadow registers. The reverse is true in supervisor mode.	

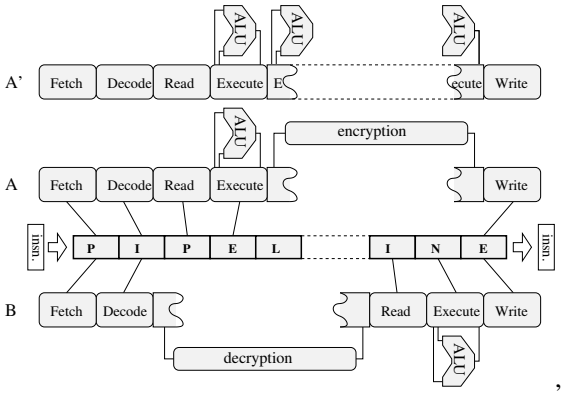

Figure 2. The pipeline is configured in different ways, 'A' and 'B', for different kinds of user mode instructions.

In [5] the protocol (*) plus the invariant is proved^{A1} to guarantee (2.11) that supervisor mode never sees in unencrypted form data that originated (encrypted) in user mode, and vice versa. It is also proved in [5] to guarantee user data is stored encrypted in memory.

4.4 **Multiuser**. Changing the encryption key signals a change of user and empties the shadow registers, so one user cannot gain access to another's unencrypted data in registers, but in any case the argument in 2.10 says that access is not be an issue in itself and the instruction set is the actual danger (fixed in Section 6).

4.5 **Further modifications** to conventional design include an address *translation look-aside buffer* (TLB) in two parts. The conventional TLB is now a back-end that remaps addresses page-wise, and a new front-end maps individual encrypted addresses to a physically backed range in first-come, first-served order. As 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 though encrypted addresses are spread randomly over the whole cipherspace. The TLB front-end will eventually be limiting, but it does not affect programs whose footprint is designed to fit in cache.

5. Performance

The original Or1ksim OpenRISC test suite codes (written mostly in assembler) established benchmarks for early prototypes, when no or very rudimentary code (C) compilation was available. Most modern performance suites still cannot be compiled because they rely on support such as linear programming and math floating point libraries, as well as system support such as 'printf'. If those could be ported in good time, debugging would take months (the original OpenRISC gcc compiler has 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 less evolved benchmarks are running, in particular Dhrystone v2.1.

TABLE 1. BASELINE V. OPTIMISED PERFORMANCE WITH 64-BIT RC2 ENCRYPTION, OR1KSIM 'ADD TEST': % FINISHING PER CYCLE.

RC2 cycles		296368		237463	
222006 instructions	mode	user	super	user	super
arithmetic	register instructions	0.2%	0.2%	0.3%	0.2%
	immediate instructions	7.8%	9.8%	9.6%	12.1%
memory	load instructions	1.0%	3.0%	1.2%	3.7%
	store instructions	1.0%	0.0%	1.2%	0.0%
control	branch instructions	1.1%	5.2%	1.3%	6.4%
	jump instructions	1.2%	5.1%	1.5%	6.3%
	sys/trap instructions	0.5%	0.0%	0.7%	0.0%
no-op instructions		7.3%	16.8%	4.4%	20.8%
prefix instructions		11.8%	—	5.5%	—
move from/to SPR instructions		0.1%	2.8%	0.1%	3.5%
wait states		20.7%	4.4%	18.6%	2.4%
(stalls)		(17.4%)	(3.7%)	(7.4%)	(0.0%)
(refills)		(3.3%)	(0.7%)	(11.2%)	(2.4%)
total		52.7%	47.3%	44.5%	55.5%

Branch Prediction (18547 tot.)		
right/wrong	✓	×
hit 55%	44%	10%
miss 44%	35%	9%

User Data Cache (2933 tot.)		
	hit	miss
read	100.0%	0.0%
write	99.7%	0.3%

User Mode Crypto.	
Encryptions	639
Decryptions	12326

Table 1 shows baseline performance (red) in the *instruction set add test* of the suite, with RC2 64-bit symmetric encryption, repeating the 2016 test in [5] so progress can be seen. The 64:16:20 mix for arithmetic:load/store:control instructions (no-ops and prefixes ignored) is close to the 60:28:12 mix in the standard textbook [20]. At the time of the 2016 test, the program spent 54.8% of the time in user mode, and 52.7% now, which is $2.1/54.8 = 4\%$ better encrypted running. Pipeline occupation is now $1-20.7/52.7 = 60.7\%$ in encrypted mode, for 607Kips (instructions per second) with the 1 GHz clock.

The top right subtable shows that individual branch records (hits) gain little (44/10) over aggregated data (misses; 35/9). The middle right subtable shows all data is write-before-read (read hits 100%) and near all (99.7%) writes are repeats to a few (0.3%) locations. The crypto table shows that most en-/decryptions are elided via write-back caching. The raw numbers would be 2942 (store) and 25995 (load+immed).

The same test with Paillier-72 (128-bit architecture) shows worse performance, as some arithmetic is done in software (Table 2):

Table 2	add test	cycles	instructions
	RC2 (64-bit)	296368	222006
	Paillier-72	438896	226185

Paillier arithmetic takes the length of the pipeline to complete. That stalls following instructions that need the result until the instruction ahead has finished, leaving the pipeline mostly empty. The disparity is greater on multiplication, which is done in software (Table 3):

Table 3	mul. test	cycles	instructions
	RC2 (64-bit)	235037	141854
	Paillier-72	457825	193887

Performance with symmetric encryptions, but not with Paillier, is sensitive to data-forwarding along the pipeline. Turning off forwarding and instruction reordering shows 33% of processor speed is due to forwarding, while reordering gives another 3% (Table 4):

Table 4	add test RC2 (64-bit) cycles		forwarding ✓	forwarding ×
	reordering	✓	296368	412062
		×	315640	441550

Paillier's insensitivity is expected because arithmetic results are not available before the penultimate stage of the pipeline. A work-around may be to create hyperthreaded programs, so instructions from an independent thread may overtake a stalled instruction.

Since the 2016 account in [4] three solutions tailored to the architecture and the bottlenecks noted above have been implemented: (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.

'Flexible staging' (a) drops cycle count from 296368 to 259349 cycles and then innovations (b), (c) contribute as follows (Table 5):

Table 5	add test RC2 (64-bit) cycles		deprefixing (b) ✓	deprefixing (b) ×
	branch both (c)	✓	237463	257425
		×	241992	259349

Branching both ways was ineffective because only 3717 branches mis-predicted, but harder-to-predict code benefits.

Those RC2-64 tables can also provide approximate numbers for AES-128 via the following Dhrystone v2.1 benchmarks (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

By that measure, the AES-128 prototype is running as a 433 MHz classic Pentium, or 250 MHz Pentium M.[8] The results are compiler-sensitive, as shown by variation through optimisation levels O0-O6 for the Pentium M in Table 6, and

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

8. See Dhrystones table at http://www.roylongbottom.org.uk/dhrystone\%20results.htm.

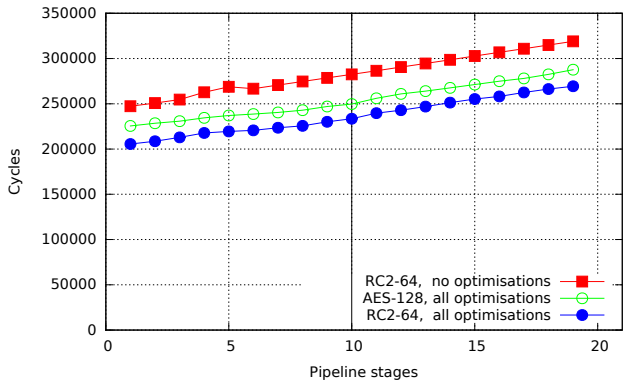

Figure 3. Number of executed cycles with symmetric encryption for the 'add test' of Table 1 (red/blue) against number of stages occupied by the encryption/decryption unit ('codec').

TABLE 7. FXA MACHINE CODE INSTRUCTIONS FOR ENCRYPTED RUNNING.

	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: r is a register index or memory location, k is a 32-bit integer, j is an instruction address increment, '$\leftarrow$' is assignment. The function $[\cdot]_{\mathscr{E}}$ represents encryption, $[\cdot]_{\mathscr{D}}$ decryption.

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 carrying immediate data. That illustrates that compilers for encrypted instruction sets must avoid inline data in instructions. The RC2 prototype equates to a 433 MHz classic Pentium, 266 MHz Pentium M.

The results may be extrapolated as required to more pipeline stages: Fig. 3 shows that each stage costs 3.1% in the baseline, but 1.7% with hardware optimisation.

6. FxA Instruction Set

Standard instruction sets are insecure for encrypted working (recall the chosen instruction attack of 2.6), but the minimal 'one instruction' HEROIC instruction set is immune to the problem.

Denote by a *fused anything and add* (FxA) instruction set one where arithmetic instructions add constants $-k_1$, $-k_2$ to operands x_1, x_2 and adds a constant k_0 to the result. So FxA multiplication does:

$$x_0 \leftarrow (x_1 - k_1) * (x_2 - k_2) + k_0$$

TABLE 8. RUNTIME TRACE FOR ACKERMANN(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
```

An FxA instruction set for encrypted working is shown in Table 7. Some instructions, e.g. addition, need only *one* constant, as

$$(x_1 - k_1) + (x_2 - k_2) + k_0 = x_1 + x_2 + (k_0 - k_1 - k_2)$$

HEROIC's instructions are a (tiny) subset. The processor enforces *no collisions between (i) encrypted constants that appear in instructions and (ii) runtime encrypted data values in registers or memory*. The implementation introduces different types of padding/blinding factors for (i), (ii). 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^{A2} depends on the operator not being able to interpret anything from changes in encrypted data or instruction constants. However, HEROIC's one-to-one encryption maps collisions to equalities underneath the encryption, invalidating the assumption. The objection is met by an *obfuscating compiler* [7] that itself varies the runtime data under the encryption.

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 register or memory location l at any given point in the trace is uniformly $1/2^{32}$.*

'The (obfuscating) compiler did it' is a valid cover for runtime cipherspace collisions. The compiler uses constants in FxA instructions to vary the runtime data at location l by a different offset each time the source code is recompiled.^{A3}

For example, the paradigmatic Ackermann function [33] 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 instead been compiled with random-looking embedded constants (the decrypted form is shown in the table, with E[-] indicating encryption). The runtime trace also shows

(encrypted) random-looking data values are written to registers before the return value (encrypted) 13 is written. Fact 2 formally implies[A4] *semantic security* of runtime data from the operator [14]. I.e., no attack does better than guessing.

It is planned to accept FxA instructions via a pre-decode stage that splits them into OpenRISC instructions.

7. Conclusion

This paper aims to communicate to the secure hardware community that encrypted working in a near conventional processor is a real possibility. A simple superscalar pipelined 32-bit OpenRISC architecture is described, but the expert community should be able to apply the design principle more generally: it is that an appropriately modified arithmetic generates encrypted working.

AES-encrypted computing here has benchmarked as a 433 MHz Pentium, on a 1 GHz clock, using our compilation toolchain. The 'FxA' modified RISC instruction set is introduced here, in which every program and trace may be interpreted arbitrarily. That makes encrypted computing as safe mathematically as the encryption key is physically.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proc. 2nd Int. Work. Hard. Arch. Supp. Sec. Priv. (HASP'13)*. ACM, June 2013.

[2] A. Biryukov. Known plaintext attack. In H. C. A. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 704–705. Springer, Boston, MA, 2011.

[3] P. Breuer and J. Bowen. A Fully Homomorphic Crypto-Processor Design: Correctness of a Secret Computer. In *Proc. Int. Symp. Eng. Sec. Soft. Syst. (ESSoS'13)*, number 7781 in LNCS, pages 123–138. Springer, Feb. 2013.

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

[5] P. Breuer, J. Bowen, E. Palomar, and Z. Liu. A Practical Encrypted Microprocessor. In C. Callegari, P. Sarigiannidis, et al., editors, *Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16)*, volume 4, pages 239–250. SCITEPRESS, July 2016.

[6] P. Breuer, J. Bowen, E. Palomar, and Z. Liu. Encrypted computing: Speed, security and provable obfuscation against insiders. In *Proc. 51st Int. Conf. Sec. Tech. (ICCST'17)*, pages 1–6. IEEE, Oct. 2017.

[7] P. Breuer, J. Bowen, E. Palomar, and Z. Liu. On obfuscating compilation for encrypted computing. In P. Samarati, M. Obaidat, and E. Cabello, editors, *Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17)*, volume 6, pages 247–254. SCITEPRESS, July 2017.

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

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

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

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

[12] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Proc. 30th Ann. Int. Conf. Th. App. Crypto. Tech. (EuroCRYPT'11)*, number 6632 in LNCS, pages 129–148. Springer, 2011.

[13] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. Garay, editors, *Adv. Crypto. – Proc. 33rd Ann. Crypto. Conf. (CRYPTO'13)*, number 8042 in LNCS, pages 75–92. Springer, Aug. 18-22 2013.

[14] S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proc. 14th Ann. Symp. Th. Comp. (STOC'82)*, pages 365–377. ACM, 1982.

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

[16] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

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

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

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

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

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

[22] 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.

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

[24] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Proc. 19th Ann. Int. Crypto. Conf. (CRYPTO'99)*, Advances Crypto., pages 388–397. Springer, 1999.

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

[26] 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)*, pages 311–324. ACM, 2013.

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

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Conf. Th. App. Crypt. Tech. (EuroCRYPT'99)*, number 1592 in LNCS, pages 223–238. Springer, 1999.

[29] D. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, Jan. 1985.

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

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

[32] 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. Priv.*, pages 38–54, May 2015.

[33] Y. Sundblad. The Ackermann function. a theoretical, computational, and formula manipulative study. *BIT Num. Math.*, 11(1):107–119, Mar. 1971.

[34] N. Tsoutsos and M. Maniatakos. The HEROIC framework: Encrypted computation without shared keys. *IEEE Trans. CAD IC Syst.*, 34(6):875–888, 2015.

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

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

[37] S. Zhang, H. Li, K. Jia, Y. Dai, and L. Zhao. Efficient secure outsourcing computation of matrix multiplication in cloud computing. In *IEEE Glob. Comms. Conf. (GLOBECOM'16)*, pages 1–6, Dec. 2016.

Appendix A

In order to facilitate the referees' work, proofs of the claims in the text are sketched here. This appendix is not for publication and is provided strictly as a courtesy to, aid and convenience for referees. There is no obligation to read or consult it. Longer versions of these proofs are to be found in the cited refereed publications ('longer' does not equal better).

Please note that, in common with all scientists, early versions of our submitted papers, including this one, are archived as technical reports on our own institutions' web pages and other sites, and, particularly in the case of security-relevant papers such as this, on the IACR electronic archive http://eprint.iacr.org/ where they also have the status of technical reports.

> *The Cryptology ePrint Archive provides rapid access to recent research in cryptology. Papers have been placed here by the authors and did not undergo any refereeing process other than verifying that the work seems to be within the scope of cryptology and meets some minimal acceptance criteria and publishing conditions.*
>
> . . .
>
> ***Posting a paper to the Cryptology ePrint Archive does not prevent future or concurrent submission to any journal or conference with proceedings: the papers in the Cryptology ePrint Archive have the status of technical reports in this respect.***

The academic principle is that ***unrefereed*** work does not count.

A1. The (*) protocol

To show that the protocol (*) of Section 4 separates user and supervisor mode data, one shows that the design establishes three invariants at processor startup, and then maintains them forever via the implemented semantics of each individual instruction (this is a reprise of the argument in [5])). The idea may be understood as a design principle for each and every instruction's logic, resolving 'what should it do in this case' questions.

Begin by naming five kinds of data:

- 𝕚 32-bit unencrypted data that originated as encrypted user data; an example is the number '1', provided (encrypted) as a program constant;
- 𝔼 encrypted user data occupying 64 or 128 bits, an example is '10 . . . 2048358178690568206', a 128-bit encryption of '1';
- 𝔻 32-bit plaintext data that originated in supervisor mode, an example is the number '6', the 'fabrication stepping number' of the processor, obtained from a special register;
- 𝔸 64-bit program addresses that notionally originate in supervisor mode, an example is the (hexadecimal) number 0x7000000015E8CC48.
- * a placeholder that stands for pending decryption (𝕚) or encryption (𝔼) but physically looks like program address zero (𝔸), 0x7000000000000000.

The invariants for individual instruction semantics are as follows:

1) In supervisor mode, *real/shadow registers contain types* 𝔼/𝕚 *or* 𝔼/* *or* */𝕚 *or* 𝔻/𝔸 respectively.
2) In user mode, *real/shadow registers contain types* 𝕚/𝔼, *or* */𝔼 *or* 𝕚/* *or* 𝔸/𝔻 respectively.
3) *Memory contains* 𝔼 *or* 𝔻 *or* 𝔸 *(hence *), but not* 𝕚 .

The invariant (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. Invariants (1) and (2) are mutually maintained by the protocol (*), as it swaps real/shadow registers on mode change.

Within the processor, two 'tag bits' inaccessible to the binary interface identify the kind of data. So registers are in fact 130 or 66 bits wide.

Every instruction is designed to preserve those invariants in both processor modes. User mode addition, for example, does 𝕚/?+𝕚/?=𝕚/*, requiring type 𝕚 in both addend registers, otherwise it raises a 'range' exception. That is, user mode addition expects decrypted data, properly tagged in both 'shadow' registers (which are swapped into 'real' position in user mode), otherwise it will not work. Moreover, when it 'does not work' the semantics is such that the result register content still satisfies the invariant. 'Leaving it alone' is a valid implementation strategy, since it would have already satisfied the invariant before the instruction ran, but other implementations are possible. All that is required is that the partner value in the other register of the aliased pair either be the placeholder * or the encrypted value. That makes 'leave it alone' the easiest option on error, but a random result may be preferred. It is important for the security properties that each instruction should have an atomic action, leaving no observable trace via the binary interface of its internal states, and 'leave it alone' may give a view of that. The 2017 publication [7] states that and three more security conditions for the action of each instruction in user mode:

(1) each instruction is a black box;
(2) each instruction reads and writes encrypted data;
(3) each instruction supports adjustment via its embedded (encrypted) constants to support arbitrary given linear shifts to its inputs and outputs as they are beneath the encryption;
(4) there may be no collisions between the encrypted constants embedded in (some) instructions and the runtime encrypted data values that occur in registers and memory.

Those conditions are satisfied by the FxA instructions of , and with them the proofs of the next section become possible.
The start condition is 𝔻 in memory and real/shadow registers appropriately configured for (1). Then all invariants are maintained through every instruction execution sequence – a program trace.

A2. Proofs

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

Suppose for contradiction that the operator has a method $f(T,C){=}y$ of knowing that the output $[y]_{\mathscr{E}}$ 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 $y{\leftarrow}[x{+}k]_{\mathscr{E}}$ 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 $[x{<}K]_{\mathscr{E}}$ 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 $[K']_{\mathscr{E}}$ instead of $[K]_{\mathscr{E}}$. 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 $[y]_{\mathscr{E}}$ but $[y{+}h]_{\mathscr{E}}$, so the method fails. It does not exist.

Now suppose for contradiction that the operator builds a new program $C'{=}f(C)$ that returns $[y]_{\mathscr{E}}$. Then its constants $[k]_{\mathscr{E}}$ are found in C and its constants $[K]_{\mathscr{E}}$ 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 $[y]_{\mathscr{E}}$ 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 $x{\leftarrow}[y{+}k]_{\mathscr{E}}$. 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 $r_0 \leftarrow [(r_1{-}k_1)\,\Theta(r_2{-}k_2){+}k_0]_{\mathscr{E}}$ with $r_0 \leftarrow [(r_1{-}k'_1)\,\Theta(r_2{-}k'_2){+}k'_0]_{\mathscr{E}}$ where $k'_i = k_i + h$, $i = 0, 1, 2$. □

A3. Obfuscating compilation

Though we appreciate that a speciality in computer science is required to understand how a compiler works or even to properly read the standard kind of semi-formal presentation of its working, we feel obligated to explain something and not just insist it works.

The reader should apprehend that the compiler works to create code that at runtime deliberately always 'misses its target' by a controlled amount beneath the encryption. We refer to that amount as an 'offset'. So, for example, instead of running code doing '2+2=4' beneath the encryption, it may instead run as '7+13=19', because the compiler has arranged for offsets of 5 and 11 respectively on the inputs, and 15 on the output (all 'beneath the encryption').

To stay in control of all this, the compiler works with a database of offsets that formally has a type expressed as

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

which is to say that one may look up in the database D a register or memory location and will get back the compiler's current plan for its (32-bit) integer 'offset' at runtime. Floating point numbers are represented at runtime as 32-bit integers via the IEEE-754 standard encoding, so this technique works to handle them too. The database is an 'obfuscation scheme', also known as an 'offset scheme'.

As the compiler works through the source code, its obfuscation scheme changes. All that we really need to do is to show that (i) the scheme can be controlled as required, and then subsequently (ii) an information-theoretic argument can be made that the distribution of possible offsets is uniformly flat across compilations. That is, we can cause the obfuscation scheme at each point in the program to be arbitrarily different from recompilation to recompilation of the same source code.

That is 'within reason'. There are points in the program code (either source or object code) between which the obfuscation scheme cannot vary, because no write to any memory location or register takes place. A trivial example is the program consisting of just a 'no-op' instruction. The obfuscation scheme planned for just before the no-op must be exactly the same as that planned for just after. Loops and gotos occasion similar restrictions. And the obfuscation scheme down two branches of a conditional must end up the same where the branches join together again.

The compiler also maintains a database that maps the source code variables to register and memory locations at each point in the program, but there is nothing special about that - all compilers do that. This database has type:

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

signifying precisely that if it is give a variable (name), it returns the location in registers or memory where it is mapped to.

The whole compiler, producing FxA machine code, has type:

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

which means that it takes source code to object ('machine') code, updating the D database as it goes.

As syntactic sugar in the following semi-formal description of its working, a pair in the cross product will be written $D : s$, rather that (D, s), as it is easier to read without parentheses.

Details of the management of database L are omitted here. They are entirely standard.

A3.1 **Sequence:** The compiler works left-to-right through a source code sequence $s_1; s_2$, which formally is expressed as follows:

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

Read that as '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 in its image when loaded into memory).

A3.2 **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 will be generated randomly by the compiler, replacing the old offset $D_0 Lx$ that previously held for the data at that location.

The compiler first emits code m_1 for the expression e which at runtime puts the result in a designated temporary location **t0** with offset $\Delta_e = D_1\mathbf{t0}$. The compiler next emits code to transfer it from there to the location Lx intended for source code variable x. That code is an add instruction, as follows. We will use the abstract semantics of Table 7 rather than the machine code instruction in order to make the formal expression easier to read:

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

The **t0** subscript for the expression compiler told it to aim at location **t0** for the result of expression e. There are several registers reserved for temporary values. The rule is that the compiler may use **t1**, **t2**, ... too for workspace, nothing 'lower'. The point here is that the compiler freely selects the obfuscation scheme for source variable x at this point. The offset Δ_x is defined by the compiler.

A3.3 **Return:** Likewise, the compiler at a 'return e' from function f gets to freely select a final offset $\Delta_{f_{\mathrm{ret}}}$ for the return value. It 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 $\Delta_{f_{\mathrm{ret}}}$ 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:

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

For completeness, the offset for **v0** is also updated in D_1 to $D_1\mathbf{v0}{=}\Delta_{f_{\mathrm{ret}}}$, though there is no real point to it as the function body ends at the return and the database is used no further by the compiler (down this branch of the source code). Although the compiler is constrained to use the same offset scheme $\Delta_{f_{ret}}$ for the return value register **v0** at every return in the body of the function, it freely chose what that value would be at the start.

The remaining source code control constructs are treated much like return. For an if statement, for example, the final offsets of

every affected variable in each branch must match at the join, but what those offsets are is freely chosen by the compiler at the point where it started compiling the statement.

The offsets Δ_l at each point in the program are effectively *inputs* to the compilation from a random distribution. They are chosen with flat probability across the whole 32-bit range, which is to say, with possible entropy. There is no statistical bias in any direction as to what the offsets are.

The runtime environment (the remote encrypted computing platform, the system adminsitrator and so on) has no knowledge of the offset scheme chosen by the compiler. The remote user is the instigator of the compilation in the safety of their own home environment, and they know the obfuscation scheme.

With all this information one can now prove Fact 2.

Proof of 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 the offset Δ_y introduced by the compiler is a maximal entropy signal, and adding it in in (sic) an instruction swamps any information the instruction might have exposed.

The obfuscating C compiler project aimed at FxA instructions is open source, available from http://sf.net/p/obfusc/. It is written in Haskell. The current state is that it is lacking long long and double (floating point) types, unions, multi-dimensional arrays, some forms of array and struct initialization, computed gotos, and all labels must be declared with '__label__' (the gcc convention for locally scoped labels). It has gotos, interior functions (at least one level), global, static and automatic variables, 1-D arrays, structs, as well as the gcc statements-as-expressions extension and (limited) inline assembler. All standard C statements are compiled, including minor extensions such as gcc case ranges ('case 1...10:').

The major difficulty is that pointers must be declared as pointing *to* somewhere (a base array, earlier or higher up in scope), via 'int foo[100]; restrict foo int *fooptr', for example. Array 'foo' is the zone into which 'fooptr' points, announced with the 'restrict foo' qualifier in the declaration.

This is necessary because the compiler keeps track of one offset scheme per zone/array, and an unqualified pointer could point into any of them at runtime. The extended type system appears consistent, but we do not yet have a formal soundness proof for it ('soundness' means that if the type system says at compile-time that the pointer will point there, then it will be so).

Unfortunately, library functions that take pointers as arguments are impossible. That is because the library compilation does not know what zone in the programmer's code the pointers will point to when called. We are currently using macros instead (they are compiled in the context of the program in which they are used) and are evaluating what further steps to take.

The FxA toolchain includes, as well as the compiler, an assembler, linker and virtual machine, plus a disassembler. Referees are welcome to examine it and try it out. It was not known before attempting the project how far this kind of machine-code-level obfuscation could be taken. But apart from the strong restriction on pointers discussed above (which could be argued to improve C), it appears that almost all C code will be compilable (about a quarter of the gcc test-suite is compiling and running correctly at this point). It was suspected that gotos would be impossible, as runtime would not be able to jump from a context with one obfuscation scheme to a context with another, but it turns out that the receiving label may be equiped with a nontrivial action (a 'come from') that realigns the obfuscation scheme on the drop-through control flow path to the way it was at the point of the lto abel's delaration, while the goto establishes the same scheme.

A similar compilation technique solves a problem with side-effecting function calls, in that the caller does not know what the callee does to global variables' obfuscation schemes, so would be 'wrong' at its next attempt to access them after a call. The callee function simply takes care on return to restore the obfuscation scheme as it was at the point of its own definition, and the caller expects it to do that and aims at the same obfuscation scheme in the setup and recovery from the call.

There is some inefficiency there, but it may be optimised away in future. A larger and inescapable inefficiency derives from a single obfuscation scheme for a whole array. That means that every write into the array must either respect the existing scheme, or change it, triggering a flood of rewrites across the array – the beahaviour is reminiscent of ORAM. We have decided on the latter approach as more secure. Array initialisations are regarded as all taking place at once in order to keep to linear order complexity (a vector write instruction is needed in the FxA instruction set).

Arrays of records ('structs') are more efficient than plain arrays in that respect, because a different obfuscation scheme is maintained by the compiler for each record field, per array. That limits the flood of rewrites to the whole array after a single write to a field to just the corresponding fields of other records.

A4. Semantic security

Fact 2 provides a probabilistic setting in which 'semantic security' (no attack is more successful than chance) can be demonstrated, as follows.

Proof. 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 $\mathrm{p}(\mathrm{bit}_{C,T} = 1$ and $F(C,T) = 1) + \mathrm{p}(\mathrm{bit}_{C,T} = 0$ 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. □