Superscalar Encrypted RISC

Measure of a Secret Computer

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

ABSTRACT

Appropriately modifying the arithmetic in a processor causes data to remain in encrypted form throughout processing, providing privacy against other users and the operator. Progress with a simple prototype superscalar pipelined RISC processor design is reported here. The mathematics has now been tested in the context of a conventional processor architecture and application binary interface (OpenRISC v1.1). We are reporting cycle-accurate results for AES-128 and Paillier-72 encrypted computing on a 128-bit architecture, and RC2-64 on 64-bit. Those obtain 104-140 Dhrystone MIPS with the symmetric encryptions on a 1 GHz base clock, equivalent to a 433 MHz classic Pentium. This paper aims to alert the hardware community that encrypted computing provably secure against insiders is theoretically possible and also possibly practical. A modified RISC instruction set is proposed in which every program and trace in the processor may be interpreted arbitrarily, which formally guarantees semantic security for user data in combination with an 'obfuscating' compiler.

KEYWORDS

Encrypted Computing, Computer Security, Data Security

ACM Reference Format:
Peter T. Breuer, Jonathan P. Bowen, Esther Palomar, and Zhiming Liu. 2018. Superscalar Encrypted RISC : Measure of a Secret Computer. In *Proceedings of Fifth Workshop on Cryptography and Security in Computing Systems (CS2'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/01.234/567_8

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 [14], 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. Running the appropriate machine code instruction set, it is mathematically impossible for the operator to infer from the computation either statistically or logically what the user's encrypted data means, despite having read and write access to it and the program code (see Section 6). Those observations have opened the way to a processor that runs 'profoundly encrypted' at near conventional speeds in user mode, because in principle only one piece of stateless logic in the processor, the arithmetic logic unit, needs changing with respect to a conventional design. In supervisor mode the processor runs unencrypted, and encryption acts as a security barrier. The approach aims at supplying a secure platform for remote computing in the cloud [33], perhaps also for embedded systems in sensitive contexts, such as automobiles or uranium centrifuges.

A sequence of cycle-accurate behavioral models have been built to (i) demonstrate the principle is correct, and (ii) explore the limits. With respect to (ii), it was unknown beforehand if conventional instruction sets and processor architectures would be compatible, and now that is confirmed, it is clear that not every program can run encrypted – 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 to prevent a *known plaintext attack* (KPA) on encrypted address sequences. The largest application suite[1] ported so far is 22,000 lines of C, but it and every other application ported (now about fifty), has worked well, surprising the authors.

The accurate models have provided good metrics and the measures are reported here. The Dhrystone v2.1 benchmark shows 104-140 MIPS running encrypted, 320-433 MHz classic Pentium speed. But the paper's objective is to summarise the state of knowledge for the first time in a hardware engineering forum and convince that it does work, encouraging the community's focus in future.

The organisation of this article is as follows. Section 2 encapsulates processor design and working in bullet points for the reader. After reviewing contemporary related architectures in Section 3, security engineering considerations in putting principles into practice are described in Section 4 and performance is described in Section 5. Section 6 sets out for the first time a modified RISC [28] instruction set that makes encrypted computation secure, in combination with the processor that runs encrypted and an 'obfuscating' compiler.

2 SUMMARY OF DESIGN AND WORKING

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

2.1 **Architecture.** The basic layout, described in [15], is the classic single pipelined RISC processor of [28], 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 bits wide (it differs per model) for encrypted 32-bit or unencrypted 64-bit data.

[1] IEEE floating point test suite at www.jhauser.us/arithmetic/TestFloat.html.

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 $2^{31}\leq z$, $2^{30}\leq z, \dots$ in turn and subtracting whenever it succeeds), the value of z can be efficiently deduced. This is a *chosen instruction attack* (CIA) [29]. 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) [3]), RC2-64 [21] and Paillier-72 [26]. The last is an additively homomorphic[2] 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.[3] 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.[4]

The choice of trialled 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 required 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 [32] (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 files. The source codes are at sf.net/p/or1k64kpu-gcc and sf.net/p/or1k64kpu-binutils. Only the assembler needs 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 [23] or introduced via a Diffie-Hellman circuit [2] 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 [16] guarantees that data originating in user mode can never be seen in unencrypted form in supervisor mode, and conversely (see 4.3^A).

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

[3]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 [19]), so this is realistic.

3 RELATED PLATFORMS

HEROIC [32] is the most comparable contemporary platform, running a 16-bit machine in Paillier-2048 encryption [26] 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 [11]. 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,[4] 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 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** [4] 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., [13]) and success means doing it as well as Ascend does. Otherwise side-channels such as cache-hit statistics [34] and power drain [22] can leak information. Ascend runs RISC MIPS instructions [27] and slows down by 12-13.5× in encrypted mode with AES-128 (absolute speeds are not given in [4]), 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 [12], not encrypted computing.

SGX machines are used [30] 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 [9].

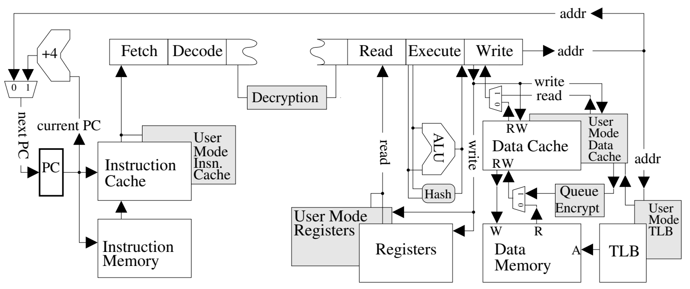

Figure 1: User mode shadow units (shaded).

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 [5] 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 [6] 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 [7] 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' [20] 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 [34] are not available.

3.5 **Oblivious RAM** (ORAM) [25] and its evolutions [24] is often cited as a defense against dynamic memory snooping. That is in contrast to static snooping, so-called 'cold boot' attacks [10] – 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 [15] in 2016 and are summarised below in 4.1, 4.2.

[4]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 $r^{\phi n} = (1{+}kn)^n{=}1{+}kn^2{+}\dots{=}1$ mod n^2 and does not affect the decrypted value. HEROIC's one-to-one encryption does not use different blinding factors.

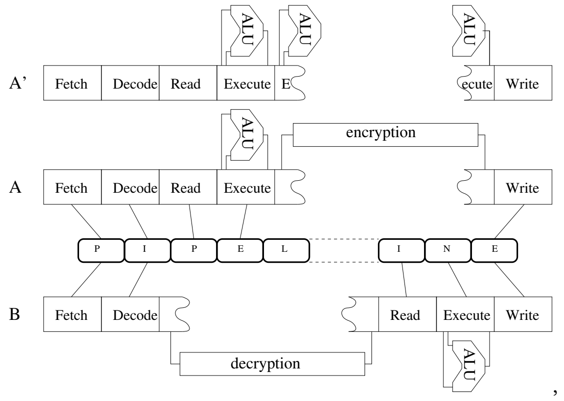

Figure 2: The pipeline is configured in different ways, 'A' and 'B', for different kinds of user mode instructions.

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 and invariant described in [16]. There are shadow registers/caches (Fig. 1) and

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.	

In [16] the protocol (*) plus the invariant is proved^{A} 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 [16] to guarantee user data is stored encrypted in memory.

The upshot is that user mode arithmetic as per 4.2 does not leak.

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 adress *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.

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.)		
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

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 shows baseline performance (red) in the *instruction set add test* of the suite, with RC2 64-bit symmetric encryption, repeating the 2016 test in [16] 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 [18]. 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 workaround may be to create hyperthreaded programs, so instructions from an independent thread may overtake a stalled instruction.

Since the 2016 account in [15] 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[5]; (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 is not effective here because only 3717 branches were predicted wrongly, but harder-to-predict code does benefit.

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 320 MHz classic Pentium, or 250 MHz Pentium M.[6] The results are compiler-sensitive, as shown by variation through optimisation levels O0-O6 for the Pentium M in Table 6, and our compiler is rudimentary. The slowdown for 128-bit AES over 64-bit RC2 is due to the 4, not 2, prefixes for an immediate constant in an instruction 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, or 266 MHz Pentium M.

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

[6]See Dhrystones table at http://www.roylongbottom.org.uk/dhrystone%20results.htm.

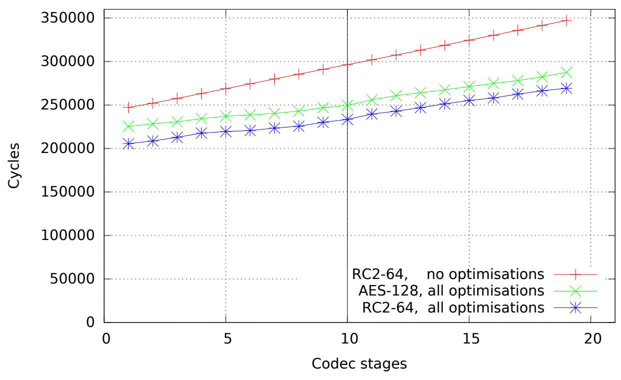

Figure 3: Number of executed cycles with symmetric encryption for the 'add test' of Table 1 (red/blue) against number of stages occupied by the encryption/decryption unit ('codec').

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.

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

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:

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

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) and checks them in the processor pipeline. Then:

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
```

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

The supporting argument^B depends on the operator not being able to read 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* [17] 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 the constants in FxA instructions to vary the runtime data written at location l by a different random offset each time the source code is recompiled.^{C}

For example, the paradigmatic Ackermann function [31] 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^D *semantic security* of runtime data from the operator [8]. That is, no attack does better than guessing.

It is planned to accommodate FxA instructions by adding a predecode stage that splits them into OpenRISC instructions on entry.

7 CONCLUSION

This paper aims to communicate to the community that encrypted working in a near conventional processor is a real possibility. A simple superscalar pipelined 32-bit OpenRISC design is described as proof of concept, but the community should be able to apply the principle more generally: it is that an appropriately modified arithmetic generates encrypted working. AES-encrypted computing here benchmarks as a 320 MHz Pentium, on a 1 GHz clock. A modified 'FxA' RISC instruction set has been introduced for which every program and trace may be interpreted arbitrarily, making encrypted computing formally as safe as the encryption key is physically.

REFERENCES

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

[2] M. Buer. 2006. CMOS-based stateless hardware security module. (April 6 2006). US Pat. App. 11/159,669.

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

[4] C.W. Fletcher, M. van Dijk, and S. Devadas. 2012. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proc. 7th Work. Scal. Trust. Comp. (STC'12).* ACM, 3–8.

[5] C. Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proc. 41st Ann. Symp. Th. Comp. (STOC'09).* ACM, 169–178.

[6] C. Gentry and S. Halevi. 2011. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Proc. 30th Ann. Int. Conf. Th. App. Crypto. Tech. (EuroCRYPT'11).* Number 6632 in LNCS. Springer, 129–148.

[7] C. Gentry, A. Sahai, and B. Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Adv. Crypto. – Proc. 33rd Ann. Crypto. Conf. (CRYPTO'13)*, R. Canetti and J.A. Garay (Eds.). Number 8042 in LNCS. Springer, 75–92.

[8] S. Goldwasser and S. Micali. 1982. Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information. In *Proc. 14th Ann. ACM Symp. Th. Comp. (STOC'82).* ACM, 365–377.

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

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

[11] D. Hardin. 2001. 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).* IEEE, 53–59.

[12] R.C. Hartman. 1993. System for seamless processing of encrypted and non-encrypted data and instructions. (1993). US Pat. 5224166.

[13] M. Hashimoto, K. Teramoto, T. Saito, K. Shirakawa, and K. Fujimoto. 2001. Tamper Resistant Microprocessor. (2001). US Pat. 0018736.

[14] P.T. Breuer and J.P. Bowen. 2013. A Fully Homomorphic Crypto-Processor Design: Correctness of a Secret Computer. In *Proc. Int. Symp. Eng. Sec. S'w. Syst. (ESSoS'13) (LNCS).* Springer, 123–138.

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

[16] P.T. Breuer, J.P. Bowen, E. Palomar, and Z. Liu. 2016. A Practical Encrypted Microprocessor. In *Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16)*, C. Callegari, P. Sarigiannidis, et al. (Eds.), Vol. 4. SCITEPRESS, 239–250.

[17] P.T. Breuer, J.P. Bowen, E. Palomar, and Z. Liu. 2017. On obfuscating compilation for encrypted computing. In *Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17)*, P. Samarati, M.S. Obaidat, and E. Cabello (Eds.), Vol. 6. SCITEPRESS, 247–254.

[18] K. Hwang. 2011. *Advanced Computer Architecture.* Tata McGraw-Hill Education. 723 pages. 2nd ed.

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

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

[21] L.R. Knudsen, V. Rijmen, R.L. Rivest, and M.J.B. Robshaw. 1998. On the Design and Security of RC2. In *Proc. 5th Int. Work. Fast S'w. Encr. (FSE'98)*, S. Vaudenay (Ed.). Springer, 206–221.

[22] P. Kocher, J. Jaffe, and B. Jun. 1999. Differential Power Analysis. In *Proc. 19th Ann. Int. Crypto. Conf. (CRYPTO'99)*, M. Wiener (Ed.). Springer, 388–397.

[23] O. Kömmerling and M.G. Kuhn. 1999. Design Principles for Tamper-Resistant Smartcard Processors. In *Proc. USENIX Work. Smartcard Tech.* USENIX, 9–20.

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

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

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

[27] D.A. Patterson and J. Hennessy. 1994. MIPS R2000 Assembly Language. In *Computer Organization and Design: the Hardware/Software Interface.* Morgan Kaufmann, Chapter A.10.

[28] D. A. Patterson. 1985. Reduced Instruction Set Computers. *Commun. ACM* 28, 1 (Jan. 1985), 8–21.

[29] S. Rass and P. Schartner. 2016. On the Security of a Universal Cryptocomputer: The Chosen Instruction Attack. *IEEE Access* 4 (2016), 7874–7882.

[30] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In*IEEE Symp.Ṡec.Ṗriv.* 38–54.

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

[32] N.G. Tsoutsos and M. Maniatakos. 2015. The HEROIC Framework: Encrypted Computation without Shared Keys. *IEEE Trans. CAD IC Syst.* 34, 6 (2015), 875–888.

[33] M. van Dijk and A. Juels. 2010. On the Impossibility of Cryptography Alone for Privacy-Preserving Cloud Computing. *HotSec* 10 (2010), 1–8.

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

In order to comply with the anonymity protocol, proofs of the claims in the text are sketched here as identifying elements of refereed publications cited in the text have had to be blanked out in the bibliography.

This appendix is not for publication and is provided as a courtesy to, aid and convenience for referees.

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 https://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.

To avoid breaking the double-blind protocol at their end, referees should use Google with caution, as it would likely reveal the authors' names. Referees may ask second persons to do web searches on their behalf.

A 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 (this is a reprise of the argument in [16])). Begin by naming five kinds of data:

🄸 32-bit unencrypted data that originated as encrypted user data;
♒ encrypted user data occupying 64 or 128 bits;
𝔻 32-bit plaintext data that originated in supervisor mode;
𝔸 64-bit program addresses that notionally originate in supervisor mode (some addresses like return addresses for a subroutine call look like they are generated in user mode but the user has no control over them and they are 'really' created and injected by the system independently of the user – being unencrypted they cannot be read or manipulated arithmetically at all by the user);
* a placeholder that stands for pending decryption (🄸) or encryption (♒) but physically looks like program address zero (𝔸).

The invariants are as follows:

(1) In supervisor mode, *real/shadow registers contain types* ♒/🄸 *or* ♒/* *or* */🄸 *or* 𝔻/𝔸 respectively.
(2) In user mode, *real/shadow registers contain types* 🄸/♒, *or* */♒ *or* 🄸/* *or* 𝔸/𝔻 respectively.
(3) *Memory contains* ♒ *or* 𝔻*or* 𝔸.

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

Every instruction preserves those invariants in both processor modes – a design principle. User mode addition, for example, does 🄸/?+🄸/?=🄸/*, requiring type 🄸 in both addend registers, otherwise it raises a 'range' exception and leaves registers as they are. The start condition is 𝔻 in memory and real/shadow registers appropriately configured for (1). Then all invariants are maintained by the program semantics.

B PROOFS

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

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

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

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

C OBFUSCATING COMPILATION

The compiler works with a database

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

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

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

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

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

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

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

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

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

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

3.2 **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 is generated randomly, replacing the old offset $D_0 Lx$ that previously held for the data at that location. The compiler emits code m_1 for the expression e which puts the result in a designated temporary location **t0** with offset $\Delta_e = D_1\mathbf{t0}$. It is transferred from there to the location Lx by a following add instruction (the **zer** location dummy in the add instruction means that field does not contribute):

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

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

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

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

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

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

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

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

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

D SEMANTIC SECURITY

Fact 2 provides the probabilistic setting for semantic security as follows. Consider a probabilistic method F that guesses for a particular runtime value beneath the encryption 'the top bit is 1, not 0', with probability $p_{C,T}$ for program C with trace T. By Fact 2, 1 and 0 are equally likely across all possible compilations C, and the probability F is right is $\text{p}(\text{bit}_{C,T} = 1 \text{ and } F(C,T) = 1) + \text{p}(\text{bit}_{C,T} = 0 \text{ and } F(C,T) = 0)$. Splitting the conjunctions, that is $\text{p}(\text{bit}_{C,T} = 1)\text{p}(F(C,T) = 1 \mid \text{bit}_{C,T} = 1) + \text{p}(\text{bit}_{C,T} = 0)\text{p}(F(C,T) = 0 \mid \text{bit}_{C,T} = 0)$. 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 $\text{p}(F(C,T) = 1 \mid \text{bit}_{C,T} = 1) = \text{p}(F(C,T) = 1) = p_{C,T}$ and $\text{p}(F(C,T) = 0 \mid \text{bit}_{C,T} = 0) = \text{p}(F(C,T) = 0) = 1 - p_{C,T}$, 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.