Fully encrypted high-speed microprocessor architecture: the secret computer in simulation

Peter T. Breuer*

Hecusys LLC,
P.O. Box 19124 Atlanta, GA, 31126, USA
Email: ptb@hecusys.com
*Corresponding author

Jonathan P. Bowen

School of Engineering,
London South Bank University,
Borough Road, London SE1 0AA, UK
Email: jonathan.bowen@lsbu.ac.uk

Abstract: The architecture of an encrypted high-performance microprocessor designed on the principle that a nonstandard arithmetic generates encrypted processor states is described here. Data in registers, in memory and on buses exists in encrypted form. Any block encryption is feasible, in principle. The processor is (initially) intended for cloud-based remote computation. An encrypted version of the standard OpenRISC instruction set is understood by the processor. It is proved here, for programs written in a minimal subset of instructions, that the platform is secure against 'Iago' attacks by the privileged operator or a subverted operating system, which cannot decrypt the program output, nor change the program's output to a particular value of their choosing. Performance measures from cycle-accurate behavioural simulation of the platform are given for 64-bit RC2 (symmetric, keyed) and 72-bit Paillier (asymmetric, additively homomorphic, no key in-processor) encryptions. Measurements are centred on a nominal 1 GHz clock with 3 ns cache and 15 ns memory latency, which is conservative with respect to available technology.

Keywords: computer security; encrypted computing.

Reference to this paper should be made as follows: Breuer, P.T. and Bowen, J.P. (xxxx) 'Fully encrypted high-speed microprocessor architecture: the secret computer in simulation', *Int. J. Critical Computer-Based Systems*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Peter T. Breuer is the Chief Research Officer at Hecusys LLC, working on encrypted computing. He has been Ramon y Cajal Research Professor in Telematic Engineering at the University Carlos III of Madrid, Spain, Lecturer in Software Systems in Computer Science at the University of Birmingham UK. He has held research posts at the University of Cambridge (Engineering) and University of Oxford (Computer Science). He has been a Visiting Associate Professor in Telecommunications Engineering at the Polytechnic University of Madrid, in Computer Science at FIT Florida, USA, and Visiting Research Fellow in Computer Science or Engineering at several institutions.

Jonathan P. Bowen is Chairman of Museophile Limited (founded 2002). He is an Adjunct Professor at Southwest University, Chongqing, China, and an Emeritus Professor at London South Bank University, where he established and headed the Centre for Applied Formal Methods from 2000. He co-authored *The Turing Guide*, a book on Alan Turing, published by Oxford University Press in 2017. He is a Fellow of the BCS and the Royal Society of Arts, and a Liveryman of the Worshipful Company of Information Technologists. His previous institutions include Imperial College London, Oxford University, and the University of Reading.

This paper is a revised and expanded version of a paper entitled 'A fully encrypted microprocessor: the secret computer is nearly here' presented at 1st Workshop on Safety & Security aSSurance for Critical Infrastructures Protection (S4CIP), Madrid, Spain, 23–26 May 2016.

1 Introduction

If the arithmetic embedded in a conventional processor is modified appropriately, then, given three technical provisos summarised in Section 4, the processor continues to operate correctly, but all its states are one-to-many encryptions of those obtained in an unmodified processor running the same program (Breuer and Bowen, 2013). That theory opens a path towards a processor that runs encrypted at the same speed as a conventional processor, because in principle only one piece of logic in the processor, the arithmetic logic unit, needs to be changed with respect to a standard design. Then data in registers, data and addresses on buses, etc., is automatically maintained in an encrypted form while the processor is running. Data input from and output to memory and disk and other I/O remains encrypted, if it starts out that way, and the machine acts as an 'encrypted processor.'

This paper summarises the state of research and development and gives performance measures on a single pipeline processor that works encrypted (our latest models achieve numbers equivalent to a fast classic pentium) with the aim of challenging the hardware and computer engineering community to apply the same approach with equal success in the context of more state-of-the-art computer architectures. It reports on architectural design, development and testing of the idea set out in the first paragraph of this section, via processor models run in simulation. The aim has been to achieve a processor design that, running encrypted:

1 protects user data from system and operator

2 presents a simple model for analysis of its security properties

3 allows fast enough running to satisfy users at least for non-interactive computation.

We believe that has been achieved and this paper aims to substantiate that.

Situations where the processor is intended to be deployed include, for example, processor farms performing heavy-duty computation on behalf of a remote client, modelling airflow on aircraft wings, or rendering for the motion picture industry, and other 'cloud-based computation' services. The prevalent danger in those situations is that a trusted operator in the computer room may steal confidential data (a so-called 'Iago' attack, Checkoway and Shacham, 2013), or the operating system itself may have been compromised to do so. But on a processor that 'works encrypted,' what may be stolen is always in encrypted form and viewing the data in memory, on buses, even in registers, shows only encryptions to the perfidious operator or subverted operating system, as required for 1 above. Nor is there information to be gained about the encryption via side channels such as cache-hit statistics or power consumption, because every encrypted arithmetic operation takes place in hardware, not in software, and takes exactly the same time and energy every time. Indeed formal proofs of the cryptographic security of user data in this context have since (after the submission of this paper) been obtained (see Section 3 for an account).

The precise contribution that this paper makes in relation to previous work is as follows. The paper extends the 2016 report (Breuer and Bowen, 2016), which discussed a model with a simple pipeline incorporating forwarding, branch prediction, instruction and data caching and speculative execution on a 64-bit architecture. That is extended here to a model that also has dynamic instruction reordering, simultaneous execution of both sides of a branch, multiple parallel decode stages, flexible staging of instructions (functional units are switched for use in different stages), and multiple multi-stage arithmetic units, on both 64- and 128-bit architectures.

Performance figures are reported here showing improvements of 50% over (Breuer and Bowen, 2016). The 128-bit results are also significant in respect of the Paillier-72 encryption (1999) embedding, which is the first embedding of a partially homomorphic encryption on the architecture. The conclusion is that the homomorphic property is not compatible with good performance. An encrypted arithmetic instruction takes the whole of the pipeline to complete and stalls instructions behind that depend on its result until it is done. The results with AES-128 encryption (Daemen and Rijmen, 2002) are also significant, reflecting a commercialisable configuration of the architecture, given that AES is the current US encryption standard. The extra development effort has comprised about 100,000 lines of model specification code, and results in a strikingly better comparison with single pipeline production processor platforms (Section 8), when the architecture embeds symmetric encryptions.

A second improvement over the account in Breuer and Bowen (2016) is to prove mathematically (Section 7) that the privileged operator cannot read nor meaningfully[a] write user-mode data, drawing that together with an account of the proof from Breuer et al. (2016) of a security property of the pipeline hardware protocol. That result has since been generalised to a more mathematical setting in Breuer et al. (2017b). It is described there in conjunction with randomised compilation, which avoids human biases towards small numbers that could otherwise be leveraged by an adversary for stochastically-based 'plaintext' attacks on encryption[b], but the proof was conceived here.

The structure of the paper is as follows. Section 2 summarises overall features of the processor design and working, giving bullet points for the reader to take away. After discussing related work and other background and context in Section 3, Section 4 summarises hardware and software provisos, as mentioned in the first paragraph of this section, required for the design to work. A definitive account of the processor architecture is given in Section 5. Section 6 gives more assurance on security questions in connection with the hardware, and Section 7 backs that up with a theoretical result showing that user data is formally secure under computation in this context. Section 8 discusses the performance of the processor design, as measured in experiments.

2 Reference points

This section intends to encapsulate in relatively short form several touchstone points that the reader may later refer back to.

Security model

The adversary in this context is the privileged operator of the processor, and/or any program running in supervisor mode. The operator can use any programmatic means that the processor interface allows, but may not physically take the processor apart or insert probes into it. The adversary's aim is to read and/or subvert user data in user programs running encrypted in user mode in the machine.

Hardware attacks

If the adversary also has a scanning electron microscope or similar advanced physical probes, then the processor operating with symmetric encryption is vulnerable because there is a codec (encryption/decryption device) inside the processor, along with the user keys. However, vulnerable areas would be protected in production by smart card-alike technology (Kömmerling and Kuhn, 1999) and scanning electron microscopes are hard to hide under the cameras of a modern computer centre so physical attacks are not a focus here. Even so, memory does not contain unencrypted user data, so it is not vulnerable to classical 'cold boot' (Simmons, 2011; Gruhn and Müller, 2013; Halderman et al., 2009) physical attacks (essentially, physically freezing the memory in order to retain an image of the memory contents when power is removed).

Additional hardware protections:

Hardware-based protections developed for conventional processors may generally be applied on top of this design, without prejudice. Such technologies make no issue over whether a calculation is encrypted or not, merely on there being details to be kept hidden. Randomising memory addressing to prevent repeat patterns from being seen, for example, has a long history ['oblivious RAM' (Ostrovsky and Goldreich, 1992; Ostrovsky, 1990; Goldreich and Ostrovsky, 1996; Lu and Ostrovsky, 2013) and its recent developments (Maas et al. 2013; Liu et al., 2015)] and there is nothing to stop it being used here. Note however, that a naturally effective randomisation and masking of addresses is already present, because many different encrypted numbers will be passed as addresses to the memory bus for what was meant by the programmer to be the same address, under what is effectively a one-to-many encryption, reckoning with padding under the (symmetric) encryption and/or blinding over the (homomorphic) encryption. Section 4 describes how software is compiled to deal with this. Protections against side-channel attacks such as 'moat' electronics (Kissell, 2006) to mask power fluctuations, may be applied too, but there is not much to protect in terms of encryption as encrypted arithmetic is done in hardware, always taking the same time and power. There are separate user-mode and supervisor-mode caches, and statistics are not provided, so side-channel attacks based on cache-hits (Wang and Lee, 2006; Zhang et al., 2012), are not available.

Simulation

The question of if our simulations give valid answers deserves an early response, as the hardware community in particular may be unfamiliar with the use for model-driven design and exploration (Giese et al., 2010). In the first place, it does not matter if simulations are accurate, as our results are principally related to encrypted versus unencrypted working in the same basic design (50–80% as fast), which is also the approach taken in the accounts of other contemporary designs reported in Section 3. If there are mistakes, they are the same mistakes on either side of the division sign.

Secondly, mistakes that make a design appear faster than it should be would make the comparison worse, as the slowness of encrypted working over unencrypted working becomes more exposed. Thirdly, the design is clocked at less than a quarter of the speed fabricated chips run at nowadays, so it is not pushing the envelope. Care has been taken to configure conservative latencies for memory and cache (13.5 ns and 3 ns respectively, adjustable). Fourthly, there is no room to get a pipeline simulation wrong in software – the pipeline moves instructions on by one pipeline stage at every clock tick. Instructions that need data do not move on until it is available, creating an empty space in the pipeline ahead and 'stalling' the instructions behind. The simulation would not work at all if that were miscoded, and timing is counting clock ticks. Model development has been incremental on that foundation, and we have been able to detect anomalies along the way. More than a million lines of final code are invested in the simulation software, for an estimated 25 years of total software engineering effort. The difficult code relates to the interaction between memory, cache and pipeline timings, but only programs with footprint that fits in the (8 MB, write-back) cache are reported here, and they are bug-free in the respect that data is always written before it is read. That ensures a cache hit on read. So, it suffices that cache interactions alone are correctly modelled in order for the results to be correct. RAM modelling is not a practical issue for dispute. Fifth, running unencrypted at 1 GHz, the machine is registering 199 MIPS for the (unoptimised) Dhrystones v2.1 benchmark reported in Section 8, or 0.2 MIPS/MHz. That is only half as good as ARMRISC machines a decade old, so the numbers from simulation pass sanity checking.

Sixth and lastly, simulation is specification, not validation. It identifies what engineers working at the logic gate level must achieve. A gate-level simulation on silicon (aka 'reality') will have its own numbers. Simulation drives what becomes real, not pre-empts it.

Prototyping achievements

The designs that we have run in simulation have resolved practical questions in connection with 3 of Section 1. In particular, it has proved possible to translate features of modern processor design – such as multistage pipelines (for superscalar[c] performance), feed-forward of data through the pipeline to avoid stalls for data dependencies between instructions in the pipeline, dynamic instruction reordering and flexible staging – to a processor that 'works encrypted.' That was unexpected, given the complexity of a modern processor, and the number of assumptions on the role of supervisor mode and user mode and the interactions with arithmetic that might underlie the state of the art in processor design.

For example, the answer to how interrupt handling should work has proved to be that it works well unencrypted and there is no incompatibility with encrypted running in practice; system calls, running unencrypted, are awkward for the user running encrypted, but a user program that wants 1 of Section 1 should not trust to the operating system for more than the lightest support, so library functions should be compiled-in to encrypted code. An operating system should load the encrypted executable and hand over control, and we have developed a minimalist set of interrupt handlers and system calls to support this functionality.

In the end, good performance has been achieved without compromising the property that data originating in user mode is separated from that originating in supervisor mode by encryption, as per 2 of Section 1, is proved in Breuer et al. (2016) for the hardware protocol implemented in the processor pipeline.

Instruction set and API

The question of what application programming interface (API) the processor may offer to programmers and compiler writers has been answered by modifying the standard 32-bit OpenRISC instruction set v1.1 (see http://openrisc.org for the behavioural level specification) for encrypted operation. In consequence, the expertise of existing manufacturers may be leveraged for production and there is no doubt as to the suitability as a platform for general purpose computing. The OpenRISC specification detailshundreds of exact conditions for the 220 machine code instructions, and the prototype passes the Or1ksim instruction test suite (http://opencores.org/or1k/Or1ksim) in encrypted mode. Data words are 32 bits long under the encryption, but they physically occupy 64 or 128 bits, etc., depending on the encryption. Instructions are 32 bits in length, conforming to OpenRISC. ‘Immediate’ data (i.e., embedded as part of an instruction) is encrypted (see Section 5), but the rest of the instruction is in the clear and can easily be read or rewritten.

Some minor changes to the OpenRISC specification have proven necessary – for example, the processor cannot, as specified at openrisc.org, return processor version number in both encrypted and unencrypted modes or the privileged observer would have an example of a plaintext and its corresponding ciphertext for study; nor may status flags (carry, overflow, etc.) generated in user mode be visible to supervisor mode, or signalling between the two modes would become possible, providing the equivalent situation – but the modifications have not had a significant impact overall. One modification, for example, is that there is no longer a displacement (e.g., ‘+4’) to be added to the address register included in a load or store instruction, as the standard OpenRISC instruction set allows. The compiler has to compensate with an extra +4 instruction. That penalises at runtime, but load/store sequences are likely to stall because memory/cache is slower than the processor and the extra instruction tends in practice to occupy what would otherwise have been an empty pipeline slot. The difficulty of theoretical prediction is why the benchmarks of Section 8 carry weight.

Toolchain

It has proved convenient to adapt the existing compiler chain port of Gnu ‘gcc’ (the name stands for ‘Gnu compiler collection’) v4.9.1 and ‘gas’ (‘Gnu Assembler’) v2.24.51 for OpenRISC to this processor (the modified source code is at http://sf.net/p/or1k64kpu-gcc and http://sf.net/p/or1k64kpu-binutils respectively). The compiler produces files in standard ELF ('executable and linkable format') layout (Levine, 1999), containing encrypted code and data sections. The assembler needs to know the encryption, while compiler and loader do not. It is desirable for the linker to know, but it has been done without so far.

Processor modes

The prototype can run as a coprocessor, running code in encrypted mode on demand; the OpenRISC specification at openrisc.org defines the support that is needed. However, it is more usual to run the processor normally. In supervisor mode (the privileged mode, in which there are no restrictions for access to processor registers or instructions), the processor runs standard (64-bit) OpenRISC machine code without encryption. In user mode (the unprivileged mode, in which access to registers, memory and instructions is restricted as per the OpenRISC specification), the processor operates encrypted, running (32-bit) OpenRISC machine code. Those are the only two modes specified by OpenRISC.

Encryption

There are two kinds of encryptions that the processor may operate with, respectively *symmetric* and *homomorphic* [the modern name for ciphers supporting the 'privacy homomorphisms' that Rivest famously envisaged in Rivest et al. (1978)]. The former require a key embedded in the processor and the latter do not, because with a homomorphic cipher standard arithmetic on the encrypted values induces the desired arithmetic operation on the values 'beneath' the encryption.

The encryptions in both cases are block ciphers, with a blocksize equal to a hardware word (64/128/256 bits for a symmetric encryption, and even more for a homomorphic encryption), each block encrypting a single 32-bit plaintext data word. The encryption for any given data word varies with time and sequence, in accord with padding bits under the encryption (symmetric cipher) or blinding factors over the encryption (homomorphic cipher). We have run the prototype with RC2 (Knudsen et al., 1998) 64-bit and Rijndael 128-bit symmetric encryption (the latter is the 'advanced encryption standard,' AES-128), and a Paillier 72-bit homomorphic encryption, gaining experience with both kinds.

Key management

Keys for the symmetric encryptions may eventually be embedded in each processor at manufacture, as is done in smart card technologies (Kömmerling and Kuhn, 1999) or introduced on need via a Diffie-Hellman circuit (Buer, 2006) or equivalent technology that does the loading operation in public view without revealing the key to even a privileged observer (keys are not available once inside the processor because there is no circuit to read them; they configure internal hardware functions and are not lodged in a processor register). However, we regard key management as a business question, because there are no special consequences for security in running with the wrong key in the machine: if user A runs with user B's key, user A's program will produce rubbish, as the arithmetic in the processor will be meaningless with respect to it; if user A runs user B's program with user B’s key, then the output will be encrypted for user B’s key, and the input will be required to be encrypted in user B’s key, which user A can neither supply nor understand (nor can user A understand B’s program, which is encrypted with B’s key). The same is true if one but not the other of A or B is the operator or operating system, which runs with no encryption. Indeed, the situation is at its worst when one of A and B is the privileged operator, and the other is an ordinary user, but that is precisely the situation 1 of Section 1 that the platform is intended to defend the user against. So the consequences, if any, of key mismanagement are intrinsically defended against.

The same reasoning is valid for homomorphic ciphers such as Paillier, although there are no keys in the processor: user B’s program run by user A will require user B’s encrypted inputs and produce user B’s encrypted outputs, which user A can neither generate nor understand (nor can user A understand user B’s encrypted program).

Cryptographic security

A formal proof is given in Breuer et al. (2016) that data that originates in user mode can always only be seen in encrypted form in supervisor mode, which operates unencrypted, despite its privileges. That implies that symmetric encryptions, which need keys to operate the encrypted arithmetic in the processor, work as securely here as homomorphic encryptions, which do not need keys. There is an opportunity there to eschew homomorphic encryption, which requires a very long blocksize and hence hardware word length for good cryptographic security, for shorter and faster symmetric encryptions.

A very acceptable level of stand-alone cryptographic security in the encryption is attained with symmetric encryptions at a 128-bit blocksize, in conjunction with (pseudo-) random padding. Brute force key search for AES-128 has complexity $2^{126.1}$ operations using the bi-clique attack (Bogdanov et al., 2011), which is the best known. Assuming 4 billion operations per second in a single core computer, that is approximately 10 billion years of computation on 10 billion 8-core machines. A 128-bit blocksize requires 128-bit wide registers and buses, memory paths, etc. and is not unusual for today’s technologies (up to 1,024-bit wide paths are available as industry moves towards ‘fat, slow and green’ from an earlier ‘narrow, fast and hot’ paradigm). The 64-bit RC2 encryption is currently unbroken but any 64-bit cipher repeats some encryption block in about 2^{32} blocks (16 GB of data processed; 4 s of computer time at 1 GHz), which aids attackers doing brute force search for the key (in about 2^{63} operations; one year of computation on eight 8-core computers), so it is not going to be the preferred option in a production machine.

In contrast, somewhat secure operation with the Paillier encryption (additively homomorphic; keyless operation in the machine) is not reached until at least 1,024 bits.

The question is if having a processor producing encrypted calculations with the instructions in the clear (apart from embedded constants,which are encrypted) compromises the underlying security of the encryption used. Section 7 answers that in the negative.

Physical limits

Aside from the path widths (which do not fit on current single chips), encrypted arithmetic operations at more than 1,024 bits for homomorphic ciphers are presently impractical on silicon for performance reasons: at least 32-stage or longer pipelines would be required at 1GHz, supposing, optimistically, that a 32-bit multiplication can be done in 1 ns in one stage. This paper gives performance measures for symmetric encryptions and varying numbers of pipeline stages up to a 20-stage pipeline, noting a consistent drop-off of 2.5% per extra pipeline stage, so by 32 stages the processor should be running at 1/5 of the maximum speed, hard to reconcile with goal 3 of Section 1. But there is further slowdown for homomorphic encryptions because many of the hardware optimisations that have proven effective with symmetric encryptions do not apply. The best option for speeding up homomorphic encryptions is based in program design and compilation: run multithreaded programs, such that a different thread may progress in the pipeline when one is stalled. However, that also requires hardware support via register aliasing (so that the instructions from different threads access different register sets while using the same 'names' for them) than we have not yet implemented, and the security implications definitely need study.

An extreme for comparison is represented by IBM's efforts at making practical computation using very long integer lattice-based fully (i.e., additively *and also* multiplicatively) homomorphic ciphers based on Gentry's 2009 discovery. Their single bit operations take of the order of a second each (Gentry and Halevi, 2011) on customised vector mainframes with a million-bit blocksize required (but it may be that newer fully homomorphic ciphers based on matrix addition and multiplication (Gentry et al., 2013) will eventually turn out to be more amenable).

We have preferred to follow Tsoutsos and Maniatakos (2015) in experimenting with homomorphic encryptions using the Paillier (additively) homomorphic cipher, as it is not as infeasible on one silicon chip as the Gentry-style (additive and multiplicative) homomorphic encryptions. While Tsoutsos and Maniatakos (2015) used a very insecure 16-bit blocksize and an unconventional 'one-instruction' stack-based encrypted processor architecture, our experiments with homomorphic encryptions have used a 72-bit blocksize. It is still very insecure, but not quite trivially so. Sticking at 72 bits for experimentation has allowed meaningful comparison with the 64-bit (symmetric) RC2 implementation because the 72 bits are shoehorned into the same length instructions and the compiler technology is the same. The drawback, however, is that multiplication and division had to be done in software, and hardware support is required for a large lookup table in connection with the comparison operators that may constitute vulnerability.

Simulator

The OpenRISC 'Or1ksim' simulator (http://opencores.org/or1k/Or1ksim) has been modified to run the processor prototype discussed here. It is now a cycle-accurate simulator, 800,000 lines of C code having been added over two years, through a sequence of prototypes. The source code archive and history is available at http://sf.net/p/or1ksim64kpu.

Memory

Memory in the processor design is conventional, but accesses are always 64-bit, or 128-bit, etc., matching the encrypted word, not 32-bit. Because of that, a program's memory footprint is × 2 or × 4 the unencrypted size, which has little impact nowadays. Cache requirements are similarly doubled or quadrupled so caching is less effective overall, but the focus of memory and cache stress lies elsewhere, in remapping at the fine granularity required for encrypted addressing (see Section 5), which introduces considerable overhead on cache faults. Support for individual address remapping (in the ‘translation lookaside buffer’; TLB) is the most demanding change required of the memory architecture.

3 Context and related work

This section sets out the background to the work, detailing the contributions of ourselves and others.

Historical context

Attempts at creating a processor that works with greater security against observation and tampering have regularly been made in the past. The earliest is likely the 2001 US Patent *Tamper Resistant Microprocessor* (Hashimoto et al., 2001) where Hashimoto et al. state “it should be apparent to those skilled in the art that it is possible to add [a] data encryption function to the microprocessor …” They meant that codecs could be placed on the path between media content stored encrypted in memory and the processor. While encryption on the way to/from memory is emphatically not part of our design, that does mark a historical beginning for encrypted computation.

Hashimoto et al. also segregated memory via access keys, and their work has left its mark in recent approaches such as Schuster et al.’s (2015) implementation of *MapReduce* for cloud-based query processing on Intel SGX™ (‘Software Guard eXtensions’) machines, which employs the SGX architecture’s hardware (Anati et al., 2013) to isolate the regions of memory involved to well-defined ‘enclaves.’ Encryption of the enclave is also an option with SGX.

SGX technology is comparable to our approach in that it enforces separations between different users and operator, but the mechanism is the management of keys to access the different memory enclaves. It provides assurance that is founded in the user’s trust in electronics designers getting details right rather than mathematical proof of principles. Recent successful attacks (Götzfried et al., 2017) against SGX show that trust is misplaced.[d]

Many earlier patents, Hashimoto et al.’s among them, managed keys kept semi-permanently within the processor to the same end as SGX. One of IBM’s early patents (‘System for seamless processing of encrypted and non-encrypted data and instructions’, Hartman, 1993) may have been the first to focus on data protection via encryption key management in-processor, although it was aimed at protecting digital media rather than enclaves in memory. The key to the encrypted (digitally protected) media is supplied encrypted so that it can only be read by the processor in question. The processor has had a private key embedded in it at manufacture time, and the public key is used to do the encryption of the media key. A flaw is that the keys and/or decrypted data may be exposed in memory when the processor has to perform a context switch for an interrupt handler, which entails saving processor register contents in memory temporarily.

Other coeval patents with related aims such as Digital Computer System for Executing Encrypted Programs (Hampson, 1989) and Secure Memory Management Unit which Utilises a System Processor to Perform Page Swapping (Buer and Eslinger, 1999) also were not specifically aimed at protecting user keys and data from code running with operating system privileges.

HEROIC

The only extant design really comparable in approach to that discussed in this paper is the HEROIC encrypted 16-bit stack machine using a 2048-bit Paillier cryptosystem (Tsoutsos and Maniatakos, 2015) addressing 216 data words in memory. But a stack machine in hardware is a fairly unexplored proposition for high-speed running at 2048 bits wide. There are some stack machine proposals a decade old attracted by the success of Java (Schoeberl, 2003, 2004; Hardin, 2001) and older implementations from a couple of decades before that in relation to programming languages of the time, but nothing from present-day manufacturers, while the architecture described here is fundamentally mainstream and may leverage contemporary know-how.

Like HEROIC our Paillier implementation uses a lookup table for arithmetic overflows (beyond the 16/32 bit range under the encryption) and that is a common limitation and vulnerability, while HEROIC also uses a second table for subtraction. Though HEROIC's encryption is deterministic, blinding factors (additive zeros) vary the encryption according to the computational history here.

Our contributions

In 2012 we showed formally that RISC (Patterson, 1985) machine code can be converted for encrypted running if it satisfies certain static typing constraints (Breuer and Bowen, 2012). Then in 2013 Breuer and Bowen gave the theory proving that an encrypted arithmetic amounts to encrypted running of an otherwise conventional processor. In Breuer and Bowen (2014b) showed that arguments and result of an arithmetic operation may be encrypted differently in this context, giving the cube (not triple) of the cryptographic security, and that the encryptions may be arranged with respect to the arithmetic such that the same computations are understood consistently differently by different observers. Breuer and Bowen (2014b, 2015) elaborated compilation strategies for the encrypted computing environment and in 2016 we reported first results of prototyping with a pipelined implementation based on OpenRISC, and a formal proof that the hardware protocol in the pipeline preserves the separation of user mode and supervisor mode data (Breuer et al., 2016). That is summarised in Section 6.

The contribution of this paper is described at the end of Section 1. In summary, this paper extends the 2016 report (Breuer and Bowen, 2016) on a model that had a simple pipeline with forwarding, branch prediction, instruction and data caching and speculative execution on a 64-bit architecture, to a model that also has dynamic instruction reordering, simultaneous execution of both sides of a branch, multiple parallel decode stages, flexible staging of instructions (the same functional units are switched for use in different stages), and multiple multi-stage arithmetic units, now tried on both 64- and 128-bit architectures. The results reported in Section 8 are comparable with off-the-shelf single pipeline processors.

A compiler technology for the platform has now been established, based on the open source *gcc* toolchain. At the time of the report in Breuer and Bowen (2016) a partially working assembler was available, and also a very limited compiler, such that tests were dependent on hand-written assembler. Now compilation is reliable and the largest C code ported to date is 22,000 lines.[e] Standard performance benchmarks have now been ported to the platform, leading to the comparison measures reported in Section 8. Those experiences have also demonstrated that operating system support, running unencrypted, is compatible with encrypted running.

Recent developments

In further work since this paper was submitted, we have extended the theory in Section 6, introducing in Breuer et al. (2017a) a modified RISC instruction set that covers the full range of conventional RISC instruction forms. The special feature of the modified instructions is that each embeds encrypted constants that vary the semantics.

For example, the modified 'multiplication by a constant' instruction has the form $y \leftarrow x * k_1 + k_2$, with two constants, not one. By setting k_2 appropriately, a compiler can cause y $(x - a) * k_1 + b$ to be executed, for arbitrary constants a, b. That remodels multiplication by a constant to input and output numbers that differ by a and b, respectively.

The modifications are needed because, in the context of encrypted computing, conventional instruction sets have recently been shown by Rass and Schartner (2016) to be vulnerable to a 'chosen instruction' attack. That is, an attacker can generate an encrypted 1 by forcing encrypted division of any encrypted x by itself, then construct any encrypted K via encrypted self addition $K - 1$ times over; then finally, branch on encrypted comparison with each K in turn allows any encrypted number to be deciphered.

The modifications not only put aside those instructions used explicitly in the attack, but they also formally generate arbitrarily many equally plausible numerical interpretations of the inputs to and outputs from each instruction, differing by a and b respectively, in the eyes of an attacker who does not yet know the encryption. The argument of Proposition 2 of this paper then may be extended to any program in the modified RISC instruction set (Breuer et al., 2017a).

Moreover, an 'obfuscating' compiler for the modified RISC instruction set is introduced in Breuer et al. (2017b) such that from recompilation to recompilation of the same program source, the runtime data at any point varies randomly with uniform distribution. In consequence, Proposition 2 extends to stochastic methods of attack too, and that guarantees cryptographic 'semantic security' (Hada, 2000) for data on the platform. To accommodate that advance, a pre-decode stage for the platform is being installed to translate modified RISC instructions to an OpenRISC instruction stream on the fly.

4 Conditions for correct running

Three formal conditions for the processor to run correctly, encrypted, by virtue of a changed arithmetic, are set out in Breuer and Bowen (2013) (bisimulation via the encryption relation):

a *The modified arithmetic in the processor must be a 'homomorphic image' of ordinary computer arithmetic.*

b *Encrypted programs may not combine* program addresses *(addresses of machine code instructions) with other data.*

c *Programs must be compiled either to save data addresses for reuse, or recalculate them the same way the next time.*

Condition 1 is the formal requirement on the modified arithmetic unit in the processor. When encrypted inputs x', y' corresponding to integers x, y are supplied to the unit for addition, what comes out must be an encryption z' of the 'plain' sum $z = x + y$. Both the abstract expression of that relation and a way to do it in hardware is to construct the output z' using an encryption unit $\mathcal{E}$ and decryption units $\mathcal{D}$, as illustrated in Figure 1 (key management not shown):

$$z' = \mathcal{E}\left(\mathcal{D}(x') + \mathcal{D}(y)\right) \tag{1}$$

But for the homomorphic Paillier encryption, working with arithmetic modulus m, instead the output z' of an encrypted addition is the conventional multiplicative product of the encrypted inputs x', y':

$$z' = x'y' \bmod m \tag{2}$$

That requires only multiplication modulo m in hardware. The 'homomorphic' property of the Paillier encryption is

$$\mathcal{D}(x'y' \bmod m) = \mathcal{D}(x') + \mathcal{D}(y') \bmod m \tag{3}$$

and substitute (3) in (2), using the adjunction $z' = \mathcal{E}(z)$ if $\mathcal{D}(z') = z$ and (1) is obtained.

Condition 2 has to do with addressing and software. Data addresses may be added to and subtracted from by the processor at runtime, so they must be treated just like other data, and that means they must be encrypted. *Program* addresses, however, are not calculated at runtime, and if they were encrypted, as the program counter is advanced by one instruction at each clock tick, that would give away the encryption. So they must be unencrypted, and care must be taken by programs not to mix the two kinds of address (link-loaders are ruled out from running encrypted by this restriction, but they can and do run in supervisor mode).

Condition 3 arises because many different encrypted numbers are generated at runtime for what the programmer intended as one memory address. They look different to the memory unit, which is not privy to the encryption. From the program's point of view, the same address seems to sporadically access different locations ['hardware aliasing' (Barr, 1998)]. The work-around is compiling so code saves the address of first use for later reuse (Breuer et al., 2015).

In practice, condition 3 only need hold for the reads following a write up to the next write, which is an opportunity (were it to be added to the design) for the processor hardware to vary the hardware memory mapping at each write, obtaining an effect like 'oblivious RAM' (Ostrovsky, 1990).

5 Architecture

The prototype complies with the OpenRISC 1.1 32/64 bit behavioural specification. Instructions are uniformly 32 bits in length but interpreted with the OpenRISC 32-bit semantics in user (encrypted) mode and the 64-bit semantics in (unencrypted) supervisor mode. Registers and memory words are all 64 bits or more (128 bits in our longer models) hiding 32 bits of user data beneath the encryption in a register or memory. The instruction set coverage across modes is summarised in Table 1.

Table 1 OpenRISC instruction set coverage in the prototype, per *U*ser and *S*upervisor mode

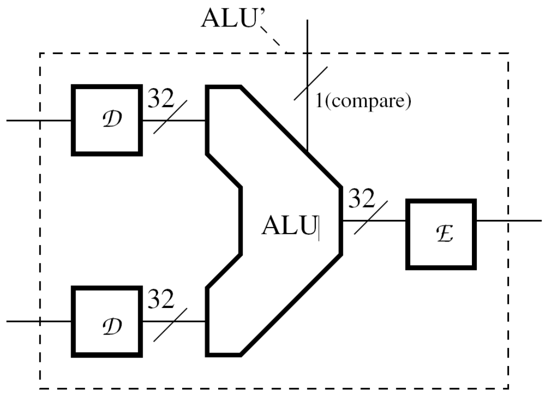
Kind\mode	*32-bit*		*64-bit*	
	Encrypted	*Unencrypted*	*Encrypted*	*Unencrypted*
Integer	U	S	–	S
Float	U	S	–	S
Vector	–	–	–	–

User mode instructions access 32 general purpose registers (GPRs) and also a few special purpose registers (SPRs). Attempts to write ‘out of bounds’ SPRs are ignored and zero is read. In supervisor mode, access is unrestricted. There is no division of memory into ‘supervisor’ and ‘user’ zones. However, as addressing is only 32-bit in user mode, supervisor mode has an intrinsically greater address range available. As per standard RISC design, memory is accessed via load and store instructions only, arithmetic is done in registers only.

In user mode, a special ‘translation lookaside buffer’ (TLB) is active. It remaps encrypted addresses, inherently scattered haphazardly over the cipherspace, to a contiguous linear sequence in a designated region on a first-come, first-served basis. The remapping database is stored in memory and cached, and the TLB hardware does lookup and assignment. Every encrypted memory address is remapped individually, costing one 128-bit TLB entry. Temporal locality of mapping assignments induces spatial locality of memory content in cache, which is effective. A TLB cache miss is very costly, invoking the minor TLB fault handler to query and update the database. Memory access *is* slower than in a conventional processor on these accounts, and more is used. There are dedicated paths for data and program memory already (‘Harvard’ layout), and we are considering adding one more for the TLB, with cache in the TLB itself.

Most of the innovations in the processor architecture are associated with the modified arithmetic for symmetric encryptions (Figure 1). In order to reduce the frequency with which the codec (represented by the encryption/decryption functions in the diagram) is brought into action, ALU operation is *extended in the time dimension*, so it covers consecutive (encrypted) arithmetic operations in user mode. Only the beginning of the series is associated with a decryption event, when encrypted data from memory or instructions is converted, and only the end of the series is associated with an encryption event. In between, arithmetic is carried out unencrypted in user mode, in ‘shadow’ registers that are unavailable in supervisor mode. That means infrequent codec use, amortising the cost. A typical test run for correctness of the arithmetic under a symmetric encryption will show for example 24,000 arithmetic operations, but only 12,000 decryption events and 600 encryption events, a 40:20:1 ratio.

Figure 1 Abstract operation with arbitrary encryption, or implementation with symmetric encryption, of a modified arithmetic logic unit for encrypted operation (ALU′ box)



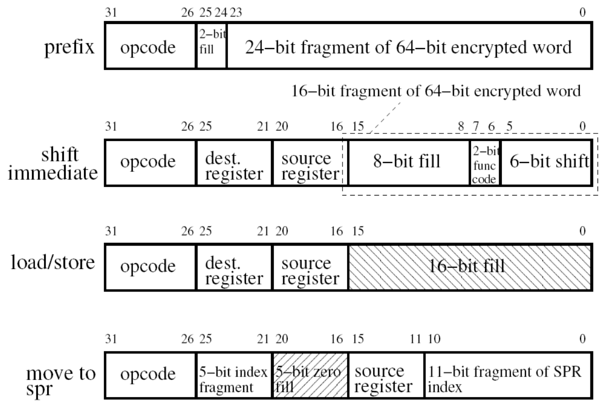
Note: Decryption units ($\mathcal{D}$) are shown on inputs and encryption unit ($\mathcal{E}$) on the output of an unmodified ALU.

An instruction in user mode sees shadow registers and an instruction in supervisor mode sees ‘real’ registers. The two are flipped via aliasing per instruction mode per stage of the processor pipeline. The hardware protocol ensures that though memory and registers may intervene as subsequent storage locations, supervisor mode never gets to see the unencrypted form of data that originates in user mode (Breuer et al., 2016, see Section 6 for a sketch of the proof).

There is just one codec embedded (over several stages) in the processor pipeline. Placing it in the pipeline is in principle very effective, because it benefits from ‘pipeline speed-up.’ A 15-stage pipeline can work on 15 instructions at a time, each in a different stage of completion, making the processor up to 15 times as fast overall. One encryption/decryption may complete per processor cycle, though each takes ten cycles overall (100 MHz encryption in contemporary hardware is achieved for AES; ten cycles at 1 GHz). The question is whether it is really effective in practice and Section 8 investigates the issue. The 40:20 ratio referred to above says that only half of arithmetic instructions invoke a decryption event, but a dependency could still stall an instruction for a full pipeline length. However, instrumentation shows that decryptions are nearly exclusively associated with arithmetic instructions that carry immediate (encrypted) data, which is decrypted in the first half of the pipeline. Since the decryption stages are staggered by the off-by-one-cycle arrival time of successive instructions, no stalls occur although only one codec is involved.

Further, decrypted instructions are cached in a user-mode-only instruction cache, so on the second encounter no decryption occurs. Gathering together the encrypted data in an instruction is also speeded up by read-ahead – 64 bits, not 32 bits, are read in parallel from instruction cache/memory per cycle to build up a ‘window’ cache 16 instructions long, 8 ahead and 8 behind in the decode stages at the front of the pipeline. Instead of needing three cycles to pick up a 32-bit instruction with (an extra 64 bits of) encrypted data for decryption, the processor usually has it all in the window cache. The same trick is worked in Hampson (1989), except that the cache is shared with supervisor mode there. The cached copy of the prefix instructions carrying the encrypted constant is changed to a no-op to prevent double replacement, and replacement is not done if the sequence crosses a cache-line boundary, to prevent a cache line flush exposing a mixed decrypted and encrypted instruction sequence.

Figure 2 Modifications to the OpenRISC instruction set for encryption



Notes: An extra ‘prefix’ instruction has been added to contain the bits of an encrypted immediate that do not fit in a following instruction, and the shift immediate instruction has been respecified to allow it to contain 16 bits of an encrypted shift that also incorporates the subfunction specifier. The displacement field in a load/store instruction is ignored, as is the designator register field in a move to/from SPR instruction.

Encrypted running has required some adjustments in the OpenRISC instruction set. To make the one codec pipeline work, instructions have been pared down (see Figure 2) to need at most one use of the codec each, on the start or end of a train of arithmetic in shadow registers. Type ‘A’ need codec use after some arithmetic, and type ‘B’ need the codec before. The pipeline is configured differently from the perspective of the two as shown in Figure 4. Instructions that do not exercise the codec, including register to register arithmetic (which run on unencrypted data in shadow registers), go through as type ‘A’ in which an early execution stage makes results available earlier. ‘A’ is also used for store and load instructions, respectively encrypting to memory from shadow and decrypting from memory to shadow registers. ‘B’ is used for ‘decrypt-first’ in instructions with embedded data.

Figure 3 Translating an assembly language instruction (left) to modified OpenRISC machine code (right)

$$\text{lws r1 4(r3)} \rightarrow \left\{ \boxed{\begin{array}{ll} \text{prefix} & [\mathcal{E}(4)]_{63-40} \\ \text{prefix} & [\mathcal{E}(4)]_{39-16} \\ \text{addi} & \text{r31 r3 } [\mathcal{E}(4)]_{15-0} \\ \text{lws} & \text{r1 0(r31)} \end{array}} \right.$$

The assembler has been modified to match the modified instruction set. In place of what would have been a single load instruction containing a displacement +4 for the address register, for example, the code emitted is as shown in Figure 3. There is no place for the

+4 in the load instruction itself so the address register is separately modified by an addition, and the encrypted +4 for the addition is supplied in two prefix instructions (64-bit encryption). Register 31 is treated non-standardly: with it as a target, no overflow or carry flag is set nor exception raised by arithmetic. OpenRISC does not have an unsigned addition and one is needed else the load intended in Figure 3 might inadvertently set unwanted flags.

Figure 4 The pipeline is configured in two different ways, 'A' and 'B', for two different kinds of user mode instructions when working with symmetric encryption

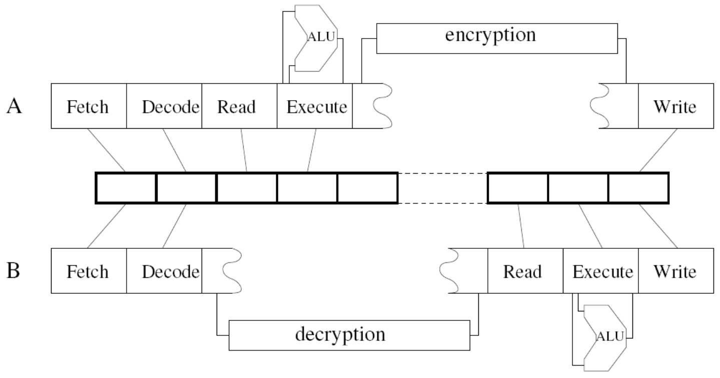
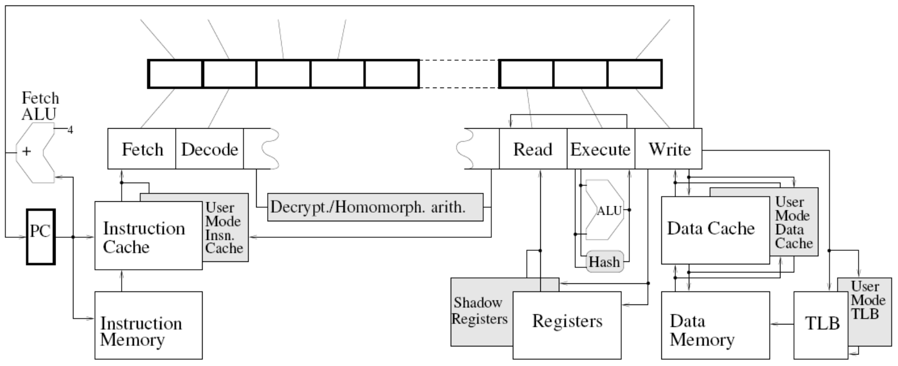


Figure 5 Pipeline integration, showing shadow units for user mode



Note: This illustration adapts drawings in Breuer et al. (2016)

Figure 5 shows the arrangement of functional units in the processor. The shadow units for user mode (encrypted) running are shown behind the corresponding units for supervisor mode (unencrypted) running.

It has turned out to be possible with symmetric encryption to pass the unencrypted data address from shadow registers to the memory unit during the processing of load and store instructions. We are nervous of the security implications, so we do not suggest that it necessarily should be done. However, the address can be hashed instead, using a hash that guarantees no collisions in 32-bit address space. The advantage is that the hardware aliasing effect discussed in Section 4 does not occur. We are currently experimenting with several fast hashes that also have good security properties.We are also experimenting with changing the hash for the address at every write (not read) to it, in order to recover more randomness in the placing of data in memory. Properly constructed code for this trope should copy the address used for write for use on subsequent reads.

The implementation for homomorphic encryptions has the same architecture as just discussed for symmetric encryptions (Figure 5) but all the instructions are of ‘A’ type and the stages devoted to the codec are devoted instead to homomorphic arithmetic operations on encrypted data. Data still passes through in the ‘shadow’ registers in user mode, but it is in encrypted form. For the Paillier homomorphic encryption, arithmetic other than addition/subtraction and comparison are implemented via software.

Some SPRs (registers accessible in supervisor mode) also have shadows, and copying to those SPRs allows interrupt handlers to save user mode shadowdata unseen for later restoration. The protocol is explained in Section 6, but there is an issue for symmetric encryptions if the user changes across the interrupt, since the second user would have the first user’s unencrypted data restored. Therefore changing keys for a symmetric encryption, which is the mark of a user change in the processor, resets all shadow registers.

6 Hardware security

As discussed in Section 1, the security goal for the processor design is to protect user information (keys, cookies, passwords, lists of weapon parts, etc.) being processed in encrypted mode against the privileged operator or operating system.

The implementation for symmetric encryptions internally decrypts data sometimes, but only in user mode, and security in the processor relies on never revealing the decrypted information to supervisor mode code. That is formally proved in Breuer et al. (2016) by the following argument on the origins of data present in registers at different times. We identify:

ℹ 32-bit unencrypted data originated as encrypted user data

♒ encrypted user data occupying 64+ bits

☽ 32-bit data in the clear that originated in supervisor mode

☾ notionally ‘decrypted’ data that originated in supervisor mode as 32-bit data and has been marked by putting a 0x7fff in the top 16 bits of the 64+ available

* a dummy ‘placeholder,’ used to indicate a pending decryption (ℹ) or encryption (♒) that looks just like the ‘decrypted’ zero datum (%).

The processor maintains the following invariants:

I1 In supervisor mode, real/shadow registers contain types ♒/ℹ or ♒/* or */ℹ or ☾/☽.

I2 In user mode, real/shadow registers will contain types ℹ/♒, or */♒ or ℹ/* or ☽/☾.

I3 *Memory contains* ♒ or ☾ or *.

Proposition 1: Unencrypted user data (type ℹ) is not exposed to the operator.

Proof: Invariants I1 and I3 guarantee the statement. The swap of real and shadow registers via aliasing on mode change maintains the invariants I1) and I2). Each instruction's implementation in each mode in the pipeline preserves these invariants [proofs are given for a small subset of instructions in Breuer et al. (2016)] and a processor reset establishes them – zeroed memory and registers will do – so by induction the invariants hold, hence the result. □

SPRs are either memory-like (no shadow register) or register-like (with shadow register) in terms of the invariants given, and almost all are not readable or writable in user mode.

The proposition and invariants tell how to securely design each instruction's semantics. Should a branch test for equality in encrypted mode work on unencrypted data too? The rules above say no, since that could be converted to an encrypted 1 or 0.

7 Data security

A simple but powerful argument is given here to show that the processor is secure in the sense that a privileged adversary who can observe everything in the processor still cannot know anything about the values of the circulating data under the encryption – if care is taken with the programming. Note that this argument has since (after submission) been extended to a general proof that user data is cryptographically secure in this context (Breuer et al., 2017b), as recounted at the end of Section 3.

Without due care, it can be easy for the operator to deduce something of what is going on. Suppose, for example, that the program is to print an integer (encrypted). Ordinarily, the operator would not know what the integer is, because it is encrypted. But suppose that it is converted to decimal digit by digit from the binary. Then the operator can see how many decimal digits are printed, allowing the magnitude to be estimated. So the careful programmer always pads to the same width with (encrypted) spaces or zeros. However, even so, the operator would see how many times the program goes through a certain loop without calculation before starting on a different section of the program, and might infer that the tight loop adds padding. So it appears that the operator ought to be able to infer information from seeing the program code running, even though the programmer believes due care has been taken. The following argument is important because it shows that it does not happen for a computationally complete class of programs on this platform.

Consider a program C that has been written using only the machine code instructions for *addition of a constant* $y \leftarrow x + k$ and branches based on *comparison with a constant* $x < K$. Those two, together with recursion, are sufficient to perform any computation, evidenced by the single combined instruction that does $x_1 \leftarrow x_1 + k_1$, $x_2 \leftarrow x_2 + k_2$, …if $x_1 < K_1$ and $x_2 < K_2$ …in the 'one instruction computer' of Tsoutsos and Maniatakos (2015) and also in Conway's well-known Fractran programming language (Maniatakos, 1987).

Proposition 2: There is no method of observation by which the privileged operator can decrypt the output of the program C above.

Proof: Suppose for contradiction that the privileged operator has some method $f(T, C)$ of knowing what the output y of the program is, although it is encrypted, having observed the trace T. Imagine, however, that every number circulating as data in the processor has, without any causation, had ‘7’ added to it under the encryption. The addition instructions $y \leftarrow x + k$ in the program still make sense, adding k to a number that is seven more than it used to be to get a number that is seven more than it used to be. The comparisons $x < K$ in the program need changing, however, because the new numbers x need to be compared with K' equal to $K + 7$ for the program to still make sense. So we modify the branches in the program to compare with K' instead of K. To the privileged operator, the new program code C' ‘looks the same,’ $C' \sim C$, because one encrypted number is as meaningful as another to the operator (we take care that there are no collisions between the encrypted values k and K used in the program code by padding or blinding appropriately, so the operator cannot tell either by means of a new collision or lack of an old one that the constants K have changed under the encryption), and the program trace T' is the same up to the encrypted numbers, which the operator cannot read, so it looks the same to the operator. That is, $T' \sim T$. As far as the operator is concerned, the inputs to the method are still the same. Those are: the structure of the program code, with everything readable except the values of constants, and the length and structure of the program trace, with everything readable except the values of the data and the constants in the executed instructions. Then the operator must see the same outcome from the method and deduce that the output under the encryption is $f(T', C') = f(T, C) = y$. Yet the output is not an encryption of y but of $y + 7$. That is, the hypothesised method does not work, so it does not exist. □

So not even the privileged operator can have a working method of cracking a user program. They are defeated because there are many possible different and consistent interpretations of the program code and runtime trace, given that the operator cannot read the encryption.

The argument may be elaborated to cover programs composed of more instructions than just the two considered, and the result may be leveraged too: for example, one may prove that the operator cannot work out what the input to the program is, because one could hypothetically alter the program to produce the original input as another output, and then the proposition applies.

Now consider again the encrypted program C written using only ‘add a constant’ $y \leftarrow x + k$ arithmetic and ‘compare with a constant’ $x < K$ branches. For definiteness suppose that the K and k come from disjoint subspaces of the cipherspace, so do not collide, and both subspaces are disjoint from that of data circulating in-processor. That can be arranged by incorporating two type bits in the padding under a symmetric encryption or in the blinding factor of a homomorphic encryption.

Proposition 3: There is no method by which the privileged operator can alter program C to produce a determined output y.

Proof: Suppose for contradiction that the operator produces a new program $C' = f(C)$ that returns (encrypted) y. Then its constants k are in C and its constants K likewise, because f has no way of arithmetically combining them (the disjoint subspaces condition means they cannot be combined arithmetically in the processor and the operator does not have the encryption key). Proposition 2 says the operator cannot read output y of C'.

In work published since the submission of this article, we have shown that the arguments above apply to the whole of an instruction set under certain conditions (Breuer et al., 2017b). Chief is that each instruction can be varied by means of embedded (encrypted) constants *a*, *b* that offset inputs *x* and outputs *y* respectively. The OpenRISC instruction $y \leftarrow x + k$ is one such, because $y \leftarrow (x - a) + k + b$ may be achieved by $y \leftarrow x + k'$, where $k' = k - a + b$. But instructions like $z \leftarrow y * z$ and even $z \leftarrow y + z$ do not satisfy the condition.Amodified OpenRISC instruction set architecture satisfying the condition is shown in Table 2, and any program *C* written in this instruction set is subject to Proposition 2 and 3.

Table 2 A modified OpenRISC machine code instruction set for working with encrypted computing

Fields	*Semantics*
add $r_0 r_1 r_2 [k]_\mathcal{E}$	Add $r_0 \leftarrow \left[[r_1]_\mathcal{D} + [r_2]_\mathcal{D} + k\right]_\mathcal{E}$
sub $r_0 r_1 r_2 [k]_\mathcal{E}$	Subtract $r_0 \leftarrow \left[[r_1]_\mathcal{D} - [r_2]_\mathcal{D} + k\right]_\mathcal{E}$
mul $r_0 r_1 r_2 [k_0]_\mathcal{E} [k_1]_\mathcal{E} [k_2]_\mathcal{E}$	Multiply $r_0 \leftarrow \left[([r_1]_\mathcal{D} - k_1) * ([r_2]_\mathcal{D} - k_2) + k_0\right]_\mathcal{E}$
div $r_0 r_1 r_2 [k_0]_\mathcal{E} [k_1]_\mathcal{E} [k_2]_\mathcal{E}$	Divide $r_0 \leftarrow \left[([r_1]_\mathcal{D} - k_1) / ([r_2]_\mathcal{D} - k_2) + k_0\right]_\mathcal{E}$
x or $r_0 r_1 r_2 [k_0]_\mathcal{E} [k_1]_\mathcal{E} [k_2]_\mathcal{E}$	Excl. or $r_0 \leftarrow \left[([r_1]_\mathcal{D} - k_1) \wedge ([r_2]_\mathcal{D} - k_2) + k_0\right]_\mathcal{E}$
…	
mov $r_0\ r_1$	Move $r_0 \leftarrow r_1$
beq $r_1 r_2\ j [k]_\mathcal{E}$	Skip *j* instructions if $[r_1]_\mathcal{D} = [r_2]_\mathcal{D} + k$
bne $r_1 r_2\ j [k]_\mathcal{E}$	Skip *j* instructions if $[r_1]_\mathcal{D} \neq [r_2]_\mathcal{D} + k$
blt $r_1 r_2\ j [k]_\mathcal{E}$	Skip *j* instructions if $[r_1]_\mathcal{D} < [r_2]_\mathcal{D} + k$
bgt $r_1 r_2\ j [k]_\mathcal{E}$	Skip *j* instructions if $[r_1]_\mathcal{D} > [r_2]_\mathcal{D} + k$
ble $r_1 r_2\ j [k]_\mathcal{E}$	Skip *j* instructions if $[r_1]_\mathcal{D} \leq [r_2]_\mathcal{D} + k$
bge $r_1 r_2\ j [k]_\mathcal{E}$	Skip *j* instructions if $[r_1]_\mathcal{D} \geq [r_2]_\mathcal{D} + k$
b *j*	Skip *j* instructions unconditionally
…	

Notes: Proposition 2 applies to arbitrary programs written in this instruction set.
Legend: the *r* are register indexes or memory locations, the *k* are 32-bit integers, the *j* are instruction address increments, '←' is assignment. The function $[\cdot]_\mathcal{E}$ represents encryption, $[\cdot]_\mathcal{D}$ decryption.

But because human beings write programs that preferentially involve small numbers such as 0, 1, 2, code and traces are always vulnerable to statistically based 'known plaintext attacks' (see, e.g., Biham and Kocher, 1994). In order to avoid that, we introduced in Breuer et al. (2017b) an 'obfuscating' compiler that from compilation to compilation of the same program varies the data values circulating at runtime via the capacity to vary the inputs and outputs of each instruction by *a*, *b* respectively. Each recompilation introduces new random and independent, uniformly distributed differences *a*, *b* from nominal at runtime, in every memory location and register, at every point of the program. In that context, programs are formally semantically secure as per Hada (2000) for user data under the encryption, against the operator as adversary.

8 Performance

The Or1ksim OpenRISC test suite codes have been compiled for encrypted running in the prototype. Table 3 details performance in the instruction set add test of the suite, with RC2 64-bit symmetric encryption. The 64:16:20 mix for arithmetic:load/store:control instructions in user mode (once no-ops and prefixes are discarded) is approximately the 60:28:12 used throughout a standard textbook (Hwang, 2011), so the results from this test are fairly typical. We have repeated exactly the 2016 test in Breuer and Bowen (2016) against our present baseline for comparison, with results in Table 3. Then, the program spent 54.8% of the time in user mode, as opposed to 52.7% now, which is a 4% (i.e., 2.1/54.8) speed-up. At the nominal 1GHz clock, pipeline occupation is now 1 – 20.7 / 52.7 = 60.7%, for 607 Kips (instructions per second). That counts no-ops and prefixes as instructions, but it serves as a rough measure.

Table 3 Baseline RC2 (64-bit symmetric encryption) performance, Or1ksim 'add test': 55% of branches are predicted, and all data is in cache

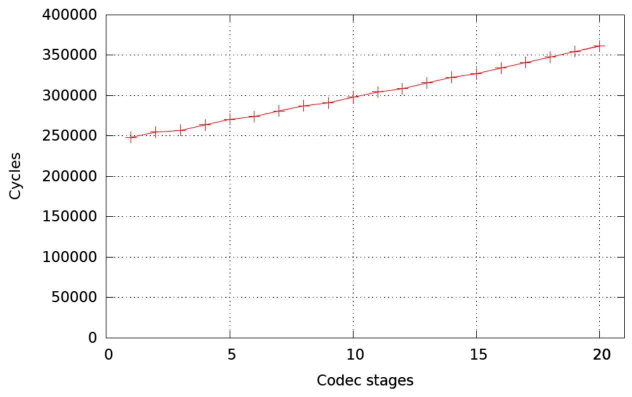
RC2@exit : cycles 296,368, instructions 222,006			
mode		*user*	*super*
arithmetic	register instructions	0.2%	0.2%
	immediate instructions	7.8%	9.8%
memory	load instructions	1.0%	3.0%
	(cached)	(1.0%)	
	store instructions	1.0%	0.0%
	(cached)	(1.0%)	
control	branch instructions	1.1%	5.2%
	jump instructions	1.2%	5.1%
	sys/trap instructions	0.5%	0.0%
No-op instructions		7.3%	16.8%
Prefix instructions		11.8%	0.0%
Move from/to SPR instructions		0.1%	2.8%
Wait states		20.7%	4.4%
(stalls)		(17.4%)	(3.7%)
(refills)		(3.3%)	(0.7%)
Total		52.7%	47.3%
Branch prediction buffer			
Hits	10,328 (55%)	Misses	8,219 (44%)
Right	8,335 (44%)	Right	6,495 (35%)
Wrong	1,993 (10%)	Wrong	1,724 (9%)
User data cache			
Read hits	2,942 (99%)	Misses	0 (0%)
Write hits	2,933 (99%)	Misses	9 (0%)

Note: The user mode pipeline is 15 stages long.

In supervisor mode, pipeline occupation is 90.6%, at 906 Kips for a 1GHz clock. One may take that as a maximum to aspire towards, though the supervisor mode instruction mix, at 50:10:40 (excluding no-ops), is low on arithmetic and heavy on control. So the encrypted user mode runs at 65.2% of the unencrypted supervisor mode. Over the entire test suite, this measure varies between 63% and 70% over tests running between 12,329 and 811,871 cycles, so 65% is a fair average.

The numbers are very sensitive to pipeline stalls and refills (which comprise 39.3% of cycles in usermode). Slightly modifying the software to eliminate them can halve runtime, while varying physical characteristics like memory latency makes little difference. The results for any one program may be extrapolated to longer codecs in the pipeline. Figure 6 shows the cost of each extra pipeline stage in the top plot ('RC2-64 No. optimisation'). Each pipeline stage costs 3.0% more cycles on the baseline 64-bit architecture.

Figure 6 Number of cycles taken to execute the 222,006 instructions of the program of Table 3 against number of stages (cycles) taken up by the codec, without reordering or hardware optimisations (a–c), and with all hardware optimisations applied. Table 3 is constructed for RC2-64 with a 10-stage codec (see online version for colours)



The same test with Paillier-72 on the 64-bit architecture shows much worse performance (Paillier does some arithmetic in software, hence the minor column 2 differences here):

Add test	*Cycles*	*Instructions*
RC2 (64-bit)	296,368	222,006
Paillier-72	438,896	226,185

The difference is mainly due to more pipeline stalls. The Paillier arithmetic always needs the full length of the pipeline to complete in, stalling following instructions that need the result as much as 11 stages behind. The disparity is even more marked on the multiplication tests where Paillier does yet more of the arithmetic in software:

Mul. test	*Cycles*	*Instructions*
RC2 (64-bit)	235,037	141,854
Paillier-72	457,825	193,887

Performance with symmetric encryption is very dependent on data-forwarding in the pipeline. The following table shows that33%of processor speed is due to forwarding, while on-the-fly instruction reordering delivers only another 3%:

Add test RC2 (64-bit) cycles		*Forwarding*	
		✓	×
Reordering	✓	296,368	412,062
	×	315,640	441,550

Codec uses (82.0% of 13,425 + 649) are chiefly for decryption of immediate data in instructions, not loads from memory (13.4%). The rest (4.6%) are for store to memory. The user mode instruction cache does treat a critical point and it halves instruction decryptions:

RC2	*Add test codec use*			
	Encryptions	*Decryptions*	*Cached*	*Total*
Store	649		2,293	2,942
Load		1,886	1,057	2,943
Immed.		11,539	11,534	23,073

Turning off the user mode instruction cache shows it is provides 10% of processor speed.

Analysis shows three remaining sources of pipeline delays here:

a *stalls* through logical dependencies between instructions;

b *prefixes* (and no-ops) occupying the pipeline for no function;

c *refills* caused by predicting the wrong branch.

In work since the 2016 account (Breuer et al., 2016) we have

a Allowed 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 to proceed and pick up the data via forwarding later.

b Doubled the fetch stage to get two instructions per cycle, filling an 8-word lookahead cache that supplies an instruction and two prefixes as one unit to decode stage.

c Introduced a secondary pipeline and speculatively execute both sides of a branch.

Flexible staging (a) takes the cycle count for the ‘add test’ of Table 3 down from 296,368 to 259,349 cycles on its own. The innovations (b) and (c) then contribute as follows:

Add test RC2 (64-bit) cycles		*Deprefixing (b)*	
		✓	×
branch both (c)	✓	237,463	257,425
	×	241,992	259,349

Figure 6 shows in the bottom plot (‘RC2-64 with optimisation’) what the costs of each extra codec stage are when all hardware optimisations are applied. Each extra stage costs 1.6% more cycles.

There are only 3,194 branch instructions issued in the 'add test', so the second pipeline has been underused. There are conflicts for the use of the arithmetic unit as follows:

ALU uses per cycle (%) [max 4, 99 times]				
0	*1*	*2*	*3*	*4*
80.2%	18.2%	1.4%	0.2%	0.0%

One extra ALU resolves 99% of these conflicts and reduces the cycle count to 232,642. The pipeline throughput in user mode is then 94,896 instruction words in 101,892 cycles, for 93.1% pipeline occupation and parity with unencrypted supervisor mode running. However, 48.6% of those instruction words are prefixes, which are artefacts of the assembler to machine code translation, illustrating that the total instruction throughput count is misleading. The meaningful count is 48,750 functional instructions executed in 101,892 cycles, for an overall efficiency of 0.48 computational operations carried out in encrypted mode per cycle. While it may be feasible to compile in such a way as to reduce the proportion of OpenRISC instructions carrying immediate constant data, which translate to the extra prefixes for encrypted mode that have lowered the efficiency here, the recent theoretical advances discussed at the end of Section 3 show that the embedded constants are important for security, so means of improving this measure rather than working around it must be found.

Although there is no space to dissect the 128-bit platform here, it works well and AES-128 runs on it, as stated in Section 1, with a 10-stage codec/14-stage pipeline. We have ported the classic Dhrystone v2.1 (25 May 1988) test by R.P. Weicker to this environment. The table below shows the results for the platform, respectively on the 64-bit architecture running the RC2 encryption, on the 128-bit architecture running the AES encryption, and on the 128-bit architecture running unencrypted in (supervisor) mode using 32-bit instructions only. The numbers are consistent with a classic single core pipeline processor running a RISC instruction set. The figure for the ARM926 (RISC) year 2000 200 MHz processor, a 5-stage single pipeline architecture, is 220 MIPS/GHz (Segars, 1998; ARM Ltd., 2000). In supervisor mode our platform also has a 5-stage pipeline, as the encryption/decryption stages are skipped.

Dhrystone v2.1	*RC2 (64-bit)*	*AES (128-bit)*	*None (32-bit)*	*Pentium M 32-bit 1 GHz*		
				O0	*O2*	*O6*
Dhrystones per second	246,913	183,486	350,877	735,294	1,470,588	2,777,777
VAX MIPS rating	140	104	199	418	836	1,580

According to the table at http://www.roylongbottom.org.uk/dhrystone%20results.htm, a PentiumM does 523 MIPS/GHz. But the results are compiler-sensitive, as shown by optimisation level O0-O6 for Pentium M, and our compiler is rudimentary. One machine instruction more can mean 2%, and subroutine frame management in OpenRISC is costly.

9 Future work

We plan to model memory bus interactions more closely to decide cache positioning and will experiment further with the dual pipelines. An 'administrator' mode will be introduced to run encrypted with privileges in support of an encrypted operating system and virtualisation. That is appropriate for 'Internet of Things' (IoT) applications in embedded devices such as cameras, where the video stream may be encrypted for viewing, but a second encryption and key supports an administrative mode that may change the viewing key.

10 Conclusions

A superscalar architecture for a microprocessor that 'works encrypted' using the OpenRISC instruction set has been described. It is based on the principle that a modified arithmetic produces encrypted processor states. Data in memory, data in registers, and data and addresses on buses exist in encrypted form, protecting user data against the operator and operating system, which runs unencrypted. It has been proved that the operator cannot read the encrypted result from a user program, and cannot modify it to produce a desired result. The simulations reported indicates that the machine works at near the speed of conventional processors.

Experience with RC2 (64-bit) and AES-128 symmetric encryptions is reported here, and with Paillier 72-bit additively homomorphic encryption. The symmetric encryptions are much more effective performance-wise. It is proved that the hardware protocol followed renders symmetric encryptions as secure or otherwise as homomorphic encryptions are, although symmetric encryption requires a key and a codec to be embedded in the processor, the latter as many stages in the processor pipeline. Conventional protections such as moat electronics, oblivious RAM, encrypting memory address and/or hashed program trace with data, etc., may be applied on top of the design. In subsequent work we have shown that the platform, in combination with a slightly modified instruction set as listed in Section 2 and a randomly 'obfuscating' compiler described elsewhere, formally provides semantic security for user data under the encryption. That means that there is no statistical or deterministic method that can determine user data from a runtime trace and the machine code program instructions. The theory, together with the speed of the platform as demonstrated here, offers practical and provable security for cloud and remote computation.

Acknowledgements

Peter Breuer wishes to thank Birmingham City University for support through a Visiting Research Fellowship in 2015 and 2016.

References

Anati, I., Gueron, S., Johnson, S.P. and Scarlata, V.R. (2013) 'Innovative technology for CPU based attestation and sealing ', in *Proc. 2nd Int. Work. Hard. Arch. Supp. Sec. Priv. (HASP'13)*, ACM, New York, NY, USA.

ARM Ltd. (2000) *Performance of the ARM9TDMI and ARM9E-S Cores Compared to the ARM7TDMI Core*, Technical report, White paper, February.

Barr, M. (1998) *Programming Embedded Systems in C and C++*, Chapter 6, 1st ed., pp.64–92, O'Reilly and Associates, Inc., Sebastopol, CA.

Biham, E. and Kocher, P.C. (1994) 'A known plaintext attack on the pkzip stream cipher', in Preneel, B. (Ed.), *Proc. 2nd Int. Work. Fast Soft. Encryption (FSE'94)*, Springer, Berlin/Heidelberg, December, No. 1008 in LNCS, pp.144–153.

Bogdanov, A., Khovratovich, D. and Rechberger, C. (2011) 'Biclique cryptanalysis of the full AES', in Lee, D.H. and Wang, X. (Eds.); *Advances in Cryptology*, pp.344–371, Springer, Berlin/Heidelberg, *Proc. 17th Int. Conf. on the Theory and Application of Cryptology and Information Security, (ASIACRYPT'11)*, 4–8 December.

Bonneau, J. and Mironov, I. (2006) 'Cache-collision timing attacks against AES', in Goubin, L. and Matsui, M. (Eds.), *Proc. 8th Int. Work. Crypto. Hard. Embedded Sys. (CHES'06)*, Springer, Berlin/Heidelberg, October, Vol. 4249 of LNCS, pp.201–215.

Breuer, P.T. and Bowen, J.P. (2012) 'Typed assembler for a RISC crypto-processor', in Barthe, G., Livshits, B. and Scandariato, R. (Eds.), *Proc. Int. Symp. Eng. Sec. Soft. Sys. (ESSoS'12)*, Springer, Berlin/Heidelberg, February, Vol. 7159 in LNCS, pp.22–29.

Breuer, P.T. and Bowen, J.P. (2013) 'A fully homomorphic crypto-processor design: correctness of a secret computer', in *Proc. Int. Symp. Eng. Sec. Soft. Sys. (ESSoS'13)*, Springer, Berlin/Heidelberg, February, No. 7781 in LNCS, pp.123–138.

Breuer, P.T. and Bowen, J.P. (2014a) 'Avoiding hardware aliasing: verifying RISC machine and assembly code for encrypted computing', in *Proc. 2nd IEEE Workshop on Reliability and Security Data Analysis (RSDA'14)*, *IEEE Int. Symp. Soft. Reliability Eng. Work. (ISSREW'14)*, IEEE Computer Society, Los Alamitos, CA, November, pp.365–370.

Breuer, P.T. and Bowen, J.P. (2014b) 'Towards a working fully homomorphic crypto-processor: Practice and the secret computer', in Jörjens, J., Pressens, F. and Bielova, N. (Eds.), *Proc. Int. Symp. Eng. Sec. Soft. Sys. (ESSoS'14)*, Springer, February, Vol. 8364 of LNCS, pp.131–140.

Breuer, P.T. and Bowen, J.P. (2016) 'A fully encrypted microprocessor: the secret computer is nearly here', *Procedia Computer Science*, Vol. 83, pp.1282–1287, *Proc. 7th Int. Conf. on Ambient Systems, Networks and Technologies (ANT'16)/6th Int. Conf. Sustainable Energy Info. Tech. (SEIT'16)/Affiliated Workshops*, April.

Breuer, P.T., Bowen, J.P. and Pickin, S.J. (2015) 'Processor rescue: safe coding for hardware aliasing', in Fujita, H. and Guizzi, G. (Eds.), *Proc. 14th Int. Conf. Intel. Soft. Meth., Tools and Tech. (SoMeT'15)*, Switzerland, Springer, September, No. 532 in CCIS, pp.137–148.

Breuer, P.T., Bowen, J.P., Palomar, E. and Liu, Z. (2016)' A practical encrypted microprocessor', in Callegari, M., van Sinderen, P., Sarigiannidis, P., Samarati, E., Cabello, L.P. and Obaidat, M.S. (Eds.), *Proc. 13th Int. Conf. Sec. Crypto. (SECRYPT'16)*, Science and Technology Publications (SCITEPRESS), Portugal, July, Vol. 4, pp.239–250.

Breuer, P.T., Bowen, J.P., Palomar, E. and Liu, Z. (2017a) 'Encrypted computing: speed, security and provable obfuscation against insiders', in Morales, A., Vera-Rodriguez, R., Lazzeretti, R., Fierrez, J. and Ortega-Garcia, J. (Eds.), *Proc. 51st Int. Carnahan Conf. Sec. Tech. (ICCST'17)*, IEEE, October, pp.1–6.

Breuer, P.T., Bowen, J.P., Palomar, E. and Liu, Z. (2017b) 'On obfuscating compilation for encrypted computing', in Samarati, P., Obaidat, M.S. and Cabello, E. (Eds.), *Proc. 14th Int. Conf. Sec. Crypto. (SECRYPT'17)*, INSTICC, SCITEPRESS, Portugal, July, pp.247–254.

Buer, M. (2006) *CMOS-Based Stateless Hardware Security Module*, US Pat. App. 11/159,669, 6 April.

Buer, M.L. and Eslinger, G.C. (1999) *Secure Memory Management Unit which Utilizes a System Processor to Perform Page Swapping*, 14 December 14, US Patent 6,003,117.

Checkoway, S. and Shacham, H. (2013) 'Iago attacks: why the system call API is a bad untrusted RPC interface', in *Proc. 18th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS'13)*, ACM, New York, NY, USA, pp.253–264.

Conway, J.H. (1987) 'Fractran: a simple universal programming language for arithmetic', in Cover, T.M. and Gopinath, B. (Eds.): *Open Problems in Communication and Computation*, pp.4–26, Springer, New York, NY, USA.

Daemen, J. and Rijmen, V. (2002) *The Design of Rijndael: AES – The Advanced Encryption Standard*, Springer, Berlin/Heidelberg.

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

Gentry, C. (2009) 'Fully homomorphic encryption using ideal lattices', in *Proc. 41st Ann. ACM Symp. Th. Comp., STOC '09*, ACM, New York, NY, pp.169–178.

Gentry, C. and Halevi, S. (2011) 'Implementing gentry's fully-homomorphic encryption scheme', in Paterson, K.G. (Ed.); *Advances in Cryptology – EUROCRYPT'11*, Vol. 6632 of LNCS, pp.129–148, Springer, Berlin/Heidelberg.

Gentry, C., Sahai, A. and Waters, B. (2013) 'Homomorphic encryption from learning with errors: conceptually-simpler, asymptoticallyfaster, attribute-based', in Canetti, R. and Garay, J.A. (Eds.); *Advances in Cryptology*, pp.75–92, Berlin/Heidelberg, Springer, *Proc. 33rd Ann. Crypto. Conf. (CRYPTO'13)*, 18–22 August.

Giese, H., Karsai, G., Lee, E.A., Rumpe, B. and Schätz, B. (Eds.) (2010) *Model-Based Engineering of Embedded Real-Time Systems*, International Dagstuhl Workshop, Dagstuhl Castle, Germany', 4–9 November 2007, Revised Selected Papers, Vol. 6100 of LNCS, Springer, Berlin/Heidelberg.

Goldreich, O. and Ostrovsky, R. (1996) 'Software protection and simulation on oblivious rams', *J. ACM (JACM)*, Vol. 43, No. 3, pp.431–473.

Götzfried, J., Eckert M., Schinzel, S. and Müller, T. (2017) 'Cache attacks on Intel SGX', in *Proc. 10th Eur. Work. Sys. Sec. (EuroSec'17)*, ACM, New York, NY, USA, pp.21–26.

Gruhn, M. and Müller, T. (2013) 'On the practicability of cold boot attacks', in *8th Int. Conf. Availability, Reliability and Sec. (ARES'13)*, September, pp.390–397.

Hada, S. (2000) 'Zero-knowledge and code obfuscation', in Okamoto, T. (Ed.), *Proc. 6th Int. Conf. Th. Applic. Crypt. Inform. Sec. (ASIACRYPT'00)*, Springer, Heidelberg/Berlin, No. 1976 in LNCS, pp.443–457.

Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J. and Felten, E.W. (2009) 'Lest we remember: cold-boot attacks on encryption keys', *Commun. ACM*, Vol. 52, No. 5, pp.91–98.

Hampson, B.E. (1989) *Digital Computer System for Executing Encrypted Programs*, 11 July, US Patent 4,847,902.

Hardin, D. (2001) 'Real-time objects on the bare metal: an efficient hardware realization of the JavaTM virtual machine', in *Proc. 4th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '01)*, IEEE Computer Society, Washington, DC, USA, pp.53–59.

Hartman, R.C. (1993) *System for Seamless Processing of Encrypted and Non-Encrypted Data and Instructions*, 29 June, US Patent 5,224,166.

Hashimoto, M., Teramoto, K., Saito, T., Shirakawa, K. and Fujimoto, K. (2001) *Tamper Resistant Microprocessor*, US Patent 2001/0018736.

Hwang, K. (2011) *Advanced Computer Architecture*, 2nd ed., McGraw-Hill Computer Science, Tata McGraw-Hill Education, India.

Johnson, M. (1991) *Superscalar Microprocessor Design*, Prentice-Hall Inc., Englewood Cliffs, NJ.

Kissell, K. (2006) *Method and Apparatus for Disassociating Power Consumed Within a Processing System with Instructions it is Executing*, US Patent App. 11/257,381, March 9.

Knudsen, L.R., Rijmen, V., Rivest, R.L. and Robshaw, M.J.B. (1998) 'On the design and security of RC2', in Vaudenay, S. (Ed.), *Proc. 5th Int. Work. Fast Soft. Encrypt. (FSE'98)*, Springer, Berlin/Heidelberg, March, pp.206–221.

Kömmerling, O. and Kuhn, M.G. (1999) 'Design principles for tamper-resistant smartcard processors', in *Smartcard '99*, May, pp.9–20.

Levine, J.R. (1999) *Linkers and Loaders*, October, Morgan Kauffman, San Francisco.

Liu, C., Harris, A., Maas, M., Hicks, M., Tiwari, M. and Shi, E. (2015) 'Ghostrider: a hardware-software system for memory trace oblivious computation', in *Proc. Int. Conf. Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS'15)*.

Lu, S. and Ostrovsky, R. (2013) 'Distributed oblivious RAM for secure two-party computation', in *Proc. Theory of Cryptography*, Springer, pp.377–396.

Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J. and Song, D. (20130 'Phantom: practical oblivious computation in a secure processor', in *Proc. ACM Conf. Comp. Commun. Sec. (SIGSAC'13)*, ACM, New York, NY, USA, pp.311–324.

Ostrovsky, R. (1990) 'Efficient computation on oblivious RAMs', in *Proc. 22nd Ann. ACM Symp. Th. Comp.*, ACM, pp.514–523.

Ostrovsky, R. and Goldreich, O. (1992) *Comprehensive Software Protection System*, US Patent 5,123,045, June 16.

Paillier, P. (1999) 'Public-key cryptosystems based on composite degree residuosity classes', in Stern, J., (Ed.): *Advances in Cryptology*, pp.223–238, Springer, Berlin/Heidelberg,, *Proc. Int. Conf. Th. Appl.Crypto. Tech. (EUROCRYPT'99)* 2–6 May.

Patterson, D.A. (1985) 'Reduced instruction set computers', *Commun. ACM*, January, Vol. 28, No. 1, pp.8–21.

Rass, S. and Schartner, P. (2016) 'On the security of a universal cryptocomputer: the chosen instruction attack', *IEEE Access*, Vol. 4, pp.7874–7882, DOI: 10.1109/ACCESS.2016.2622724.

Rivest, R.L., Adleman, L. and Dertouzos, M.L. (1978) 'On data banks and privacy homomorphisms', in DeMillo, R. et al. (Eds.): *Foundations of Secure Computation*, pp.169–179, Academia Press.

Schoeberl, M. (2003) 'JOP: a Java optimized processor', in *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'03)*, Springer, November, No. 2889 in LNCS, pp.346–359.

Schoeberl, M. (2004) 'Java technology in an FPGA', in Becker, J., Platzner, M. and Vernalde, S. (Eds.), *Proc. 14th Int. Conf. on Field-Programmable Logic and its Applications (FPL'04)*, Springer, Berlin/Heidelberg, August, pp.917–921.

Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G. and Russinovich, M. (2015) 'VC3: trustworthy data analytics in the cloud using SGX', in *IEEE Symp. Security and Privacy*, May, pp.38–54.

Segars, S. (1998) The ARM9 family-high performance microprocessors for embedded applications. In Proc. Int. Conf. Comp. Design: VLSI in Computers and Processors (ICCD'98), pages 230–235, October 1998.

Simmons, P. (2011) 'Security through amnesia: a software-based solution to the cold boot attack on disk encryption', in *Proc. 27th Ann. Comp. Sec. Appl. Conf. (ACSAC'11)*, ACM, New York, NY, pp.73–82.

Tsoutsos, N.G. and Maniatakos, M. (2015) 'The HEROIC framework: encrypted computation without shared keys', *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 34, No. 6, pp.875–888.

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

Zhang, Y., Juels, A., Reiter, M.K. and Ristenpart, T. (2012) 'Cross-VM side channels and their use to extract private keys', in *Proc. ACM Conf. Comp. Comm. Sec. (CCS'12)*, ACM, New York, NY, pp.305–316.

Notes

a 'Write meaningfully' means the operator would write the value they intend in the data under the encryption. The operator can always overwrite user data, but is not intrinsically in control of what it means when decrypted.

b A 'stochastically based' plaintext attack is still a brute force key-search, but it may abort an attempt early if it is not going to result in 0, 1, or whatever small number the 'plaintext' is hypothesised to be. It will have an advantage over pure chance because the number to be obtained from decryption will in fact be 0, 1, etc. more times than pure chance would dictate. One particular program code and trace may not yield to the attack, but on average more than chance permits will.

c 'Superscalar' means work is done in each processor core on many instructions at once in every cycle (Johnson, 1991).

d The timing attack against AES was well known Bonneau and Mironov (2006) and engineers had expected to prevent it by interfering with system timing for code running inside an SGX enclave, but the attackers directly counted instructions executed.

e The 22,000 lines refers to the IEEE floating point test suite at http://www.jhauser.us/arithmetic/TestFloat.html. Floating point arithmetic to IEEE specifications has previously been seen as impractical for encrypted computing, leading to alternative proposals (Dimitrov et al., 2016).