View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 83 (2016) 1282 – 1287
Procedia Computer Science

1st Workshop on Safety & Security aSSurance for Critical Infrastructures Protection (S4CIP 2016)

A Fully Encrypted Microprocessor

The Secret Computer is Nearly Here

Peter T. Breuer[a,*], Jonathan P. Bowen[b]

[a] *Hecusys Inc, Atlanta, GA*
[b] *London South Bank University, London, UK*

Abstract

A high-speed pre-production superscalar microprocessor that 'works encrypted' is described here. Data in registers, on buses, and (in consequence) in memory, is kept in encrypted form. It is intended to protect user data being processed remotely or overseen by untrusted operators.

© 2016 Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license
(http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Conference Program Chairs
Keywords: Computer Security, Encrypted Computing

1. Introduction

In 2013, the authors showed[1] that if the arithmetic in a conventional processor is modified appropriately, then, given three provisos (described in Section 2), 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. In particular, data in registers, data and addresses on buses, etc., is kept in an encrypted form. In consequence, the data input from and output to memory and disk or other I/O may be encrypted to start with and stays that way. Such a processor may evidently be of use where it is important to protect user data from the possibility of a dishonestly acting owner/operator, such as in voting machines[2], smart meters[3], ATMs, and in processing big data in the cloud. A follow-up paper in 2014 pointed to several possible directions[4] for a practical implementation.

Development of the idea over the last two years has resulted in the working prototype the architecture of which is described here. The prototype is sophisticated enough to bear comparison with off-the-shelf production processors: it is superscalar (it executes multiple instructions at a time), with a pipeline in which full 'forwarding' has been embedded (pipeline stalls in which an instruction behind waits for data from an instruction ahead have been eliminated as far as is logically possible), along with branch prediction, instruction and data caching, speculative execution, etc. It is intended in its present form to run as a coprocessor: just as a GPU takes on code sent to it that performs graphically-oriented calculations, so this processor takes on code sent to it that is to be run encrypted.

E-mail address: pbreuer@hecusys.com

1877-0509 © 2016 Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license
(http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2016.04.267

The technical question answered by prototyping has been whether or not an architecture based on a changed arithmetic can really attain the performance level of a modern general purpose processor, running encrypted. Modern processors break instructions into small operations and execute the parts simultaneously along one or more *pipelines*. Some of the execution is *speculative* – liable to be discarded and/or reversed – and instructions and their component parts may be accelerated or delayed according to a complex system of dependencies and constraints. A processor is not purely numerical – it is supposed to react to arithmetic overflows and a host of exception conditions and flags that might not be compatible with a modified arithmetic and encrypted execution. It is not a priori clear that any of that level of technological practice is sustainable, and the prototype has been a platform on which to test these issues, developing and refining solutions where there turn out to be problems. Broadly speaking, we do have the performance required, and we do have exactly the correct instruction semantics in all aspects with respect to a recognised standard.

The prototype runs the OpenRISC version 1.1 rev. 0 instruction set, with minor adaptations and conforms to the specification `opencores.org/or1k/Architecture_Specification`. It has the register structure and behavioural semantics described therein. The prototype passes the Or1ksim functional test suite (`opencores.org/or1k/Or1ksim`) in encrypted and unencrypted running. Data words are logically 32 bits under the encryption, but they physically occupy a whole encryption block, 64 or 128 bits, etc., as the case may be, depending on the encryption used. The prototype is set up to use 64-bit Rijndael[5] symmetric encryption and a 64-bit physical word, but it has also been run with a 72-bit Paillier encryption[6]. The latter is a (slower) asymmetric encryption with an additive property that means that no keys need be embedded in the processor, whereas for the Rijndael an embedded key is required. The Rijndael implementation, however, is the focus for this article, as we believe it to be the practical option.

The objective of the design is to permit the system operator only to be able to see user data in encrypted form, in a processor that works at near normal speed. Section 4 below reports the speed of this prototype running in encrypted mode as 60-70% of the speed in unencrypted mode, with improvements still available. An instruction is emitted every 1.67-1.8 machine cycles, and with, say, a 2GHz clock, that is $1.1\text{-}1.2\times10^9$ instructions per second. One should compare with the speed of a smart card[7], ubiquitously used today to provide secure computing solutions, standardly clocked at a 3.57MHz or 4.92MHz for approximately 10^6 instructions per instruction, hundreds of times slower.

Previous efforts at creating a processor that works with greater security against observation and tampering have concentrated on protecting memory with encryption and keyed access, while the processor itself continues to work unencrypted. The earliest work in that direction appears to be Hashimoto et al.'s US Patent for a "Tamper Resistant Microprocessor"[8] where the authors 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 an encryption/decryption device ('*codec*') could be placed on the path between entertainment media content stored encrypted in memory and the processor. In that arrangement, the user is seen as the possible attacker, trying to get at unencrypted media content, and the system is seen as the defender. That has echoes in very recent approaches such as Schuster et al.'s implementation of *MapReduce* for cloud-based query processing[9] on Intel SGX machines, which employs the machine's built-in hardware[10] to isolate the regions of memory involved to well-defined 'enclaves', and encryption may also feature. We do not focus on key management here, but many works, Hashimoto et al.'s among them, offer management of keys within the processor as the means of enforcing security barriers.

In contrast, our approach relies on the principle of having *everything* that passes through the processor in encrypted form, making the security analysis dependent on questions of computer architecture, not electronics, and we aim to protect the user from the system rather than the other way round. There are no codecs on the path from processor to memory, slowing access. The Rijndael codec is embedded (as 10 stages) in the 15-stage processor pipeline, through which every instruction must pass, so there is no *extra* delay per instruction; one is still emitted (at best) every cycle.

The layout of this paper is as follows. In Section 2 the hardware/software conditions for the design to work are laid out. An account of the architecture is given in Section 3. Section 4 discusses performance, setting out the numbers.

2. Conditions for encrypted running

There are well-defined conditions established[1] for a processor of this kind to run correctly, encrypted (it is a bisimulation of a conventional processor via the encryption relation):

(A) *The modified arithmetic implemented in the processor must be a 'homomorphic image' of ordinary computer arithmetic*;

(B) *encrypted programs must never combine* program addresses *(the addresses of machine code instructions) with other data values*;
(C) *programs must be compiled either to save data addresses for reuse, or recalculate them exactly the same way the next time*.

The first proviso (A) means that *the modified arithmetic in the processor must be a 'homomorphic image' of ordinary computer arithmetic*. That is, 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 $z = x + y$. The direct way to do that in hardware is to construct the output z' exactly as described, using an encryption device $\mathcal{E}$ and a decryption device $\mathcal{D}$:

$$z' = \mathcal{E}(\mathcal{D}(x') + \mathcal{D}(y'))$$

and the implementation for Rijndael in the prototype is analogous (see Section 3), but it is not the only possibility. For example, for the Paillier encryption modulus m, one has instead

$$z' = x'y' \mod m$$

needing only multiplication modulo m to be implemented in the hardware, with no encryption or decryption device, thanks to the 'homomorphic' property $\mathcal{D}(x'y' \mod m) = \mathcal{D}(x') + \mathcal{D}(y') \mod m$ of the Paillier encryption.

The second proviso (B) has to do with memory addressing and the kind of programs that can run. Because data addresses look no different from other numbers, and are produced dynamically in the course of a program, for example by adding an offset to a base address, inevitably *data addresses are encrypted* exactly as other data is. However, *program addresses* (addresses of program instructions) *are not encrypted*. The program counter in any processor is advanced by a constant (the length of an instruction in bytes) at each tick of the clock and that would allow an attack against the encryption, if the encryption were used for program addresses. The solution adopted is not to encrypt program addresses at all. To conform, *programs must never combine program addresses with ordinary data values*.

So link-loaders and compilers must run in supervisor mode, or remotely on the user's own platform (encrypted code for a possibly hostile platform will not sensibly do late linking).

The third proviso (C) is due to the fact that many different encryptions may be generated at runtime for what the programmer intended as one memory address. They look different to memory, which is ordinary RAM. From the program's point of view, the same address seems to sporadically access different locations ('hardware aliasing'). Code that steps in reverse down a string is problematic, for example. The right thing for the program is to save the address of each character first time for reuse.

Conforming machine code may either be generated from source or existing machine code (after encryption/translation) may be mechanically checked for conformance[11].

The GNU 'gcc' v4.9.1 compiler (`github.com/openrisc/or1k-gcc`) and 'gas' v2.24.51 assembler port (`github.com/openrisc/or1k-src/gas`) for the OpenRISC 1.1 architecture have been modified to emit code that respects these provisos[12]. The modified source code is at `sf.net/p/or1k64kpu-gcc` and `sf.net/p/or1k64kpu-binutils`.

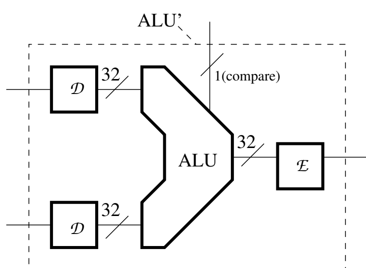

Fig. 1. Idealised modified arithmetic logic unit (ALU) for encrypted operation (ALU′ box) showing decryption devices ($\mathcal{D}$) on inputs and encryption device ($\mathcal{E}$) on output.

3. Architecture

In user mode, the prototype processor runs on encrypted data and executes the OpenRISC 32-bit instruction subset (those instructions that target 32-bit data). Encrypted data physically occupies 64 bits (or more, depending on the encryption used), but it contains only 32 bits of data when decrypted. A 64-bit instruction run in user mode raises an 'illegal instruction' exception. As per the OpenRISC specification, user mode accesses the 32 general purpose registers (GPRs), and a very few permitted special purpose registers (SPRs). Attempts to write 'out of bounds' SPRs are silently ignored in user mode, and zero is read. Running encrypted, the OpenRISC 32-bit integer and floating point instruction set coverage is complete.

In supervisor mode the processor may execute either 32- or 64-bit instructions and access to registers is unrestricted. There is no enforced division of memory into 'supervisor' and 'user' parts, so a supervisor mode process can read user data in memory, but the user data will be in encrypted form.

Instructions divide naturally into two kinds: 'immediate' instructions, which carry 16 bits of data in the (32-bit) instruction itself, and the rest, which do not. Immediate instructions are problematic in user mode because we want them to carry data in encrypted form, but encrypted data takes up 64 bits or more and an instruction is only 32 bits long, so it does not fit. To solve this problem, a *prefix* instruction has been added to the instruction set. An immediate instruction is preceded in the instruction stream by prefix instructions, carrying the initial segments of the encrypted datum, and the immediate instruction itself carries only the final 16-bit segment. Those OpenRISC immediate instructions that are supposed to carry fewer than 16 bits of data (register shifts and rotations each carry 5 or 6 bits) have been respecified to carry exactly 16 bits of data, 10 bits of which are discarded in unencrypted running.

The instruction pipeline in (unencrypted) supervisor mode is the standard short 5-stage fetch, decode, read, execute, write pipeline expected of a RISC processor[13], except that it is physically embedded in a longer pipeline that is traversed in full by (encrypted) user mode instructions. The pipeline is configured in two different ways for the user mode instructions as shown in Fig. 2 (the hardware for those stages with two different configurations is doubled). The reason is that, in order to reduce the frequency with which codecs are brought into action for user mode instructions, ALU operation has been effectively *extended in the time dimension*, so that it covers a series of consecutive (encrypted) arithmetic operations in user mode. Only the first of the series is associated with a decryption event and only the last of the series is associated with an encryption event. Longer series mean less frequent codec use. The two pipeline configurations described below cover the needs of instruction processing in encrypted running.

The 'A' configuration is deployed when a store instruction puts an encrypted result into memory, or a load instruction decrypts incoming data from memory. The 'B' configuration is used when encrypted immediate data in an 'add immediate' instruction is read in. Instructions that do not exercise the codec pass through the pipeline in 'A' configuration, because the early execution makes results available for early forwarding to instructions entering behind, avoiding pipeline stalls. The codec covers 10 stages in the prototype implementation, corresponding to 10 clock cycles per encryption/decryption, but that may be varied to suit the encryption.

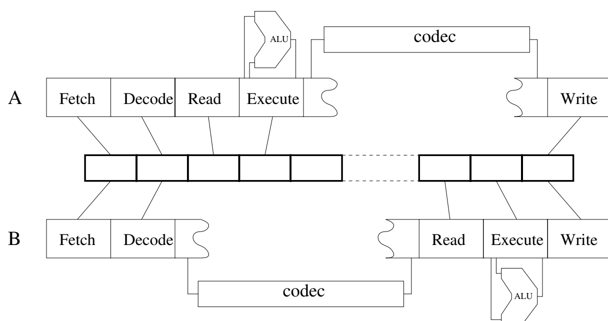

Fig. 2. The pipeline is configured in two different ways, 'A' and 'B', for two different kinds of user mode instructions during encrypted working.

To support this mode of operation, there is a private set of user-mode-only registers that shadow the GPRs (and the few SPRs accessible in user mode). These contain the decrypted version of the encrypted data in the 'real' GPRs and SPRs. They are aliased in as registers for read and write stage of a user mode instruction, and aliased out for supervisor mode instructions, so they are unavailable to supervisor mode. The protocol maintains the register entries in decrypted form in the shadow registers from one instruction to the next in user mode.

Additionally, a small user-mode-only data cache retains the unencrypted version of any encrypted data that is written to memory during user mode operation. On load from memory, this cache is checked first. Almost all execution stack reads in normal operation are intercepted by this mechanism. The cache is physically within the processor boundary, so will be covered by the measures that protect the processor chip from spying or interference (e.g., those that accrue from a smart card-like fabrication and layout).

Note that program addresses are unencrypted (as opposed to data addresses, which are encrypted), which potentially is a source of confusion in user mode. A peculiar protocol addresses the issue: unencrypted 32-bit addresses zero-filled to 64 bits are regarded as an 'encrypted' form, and they are 'decrypted' to an 'unencrypted' form consisting of the same data with the top 16 bits of 64 rewritten to 0x7fff. Thus an instruction such as jump-and-link (JAL) in user mode, which fills the return address (RA) register with the program address of the next instruction, writes the

zero-filled address to the real RA register, and the 0x7ff form to the shadow RA register. The padding under the encryption is arranged so that (really) encrypted data avoids both forms of program address.

In principle, encrypted addresses emanating from the prototype fall anywhere in the full address range (although the addresses under the encryption are 32-bit). Since no real machine ever has even a full 64 bits-worth of RAM, address translation conventionally takes place within the memory management unit to a physically backed area of memory via a 'translation look-aside buffer' (TLB). However, the TLB is conventionally organised at page-sized granularities, saying where each 8KB-sized area of logical addressing should be translated to in physical addressing terms. That architecture is not appropriate here because encrypted addresses are not clustered, if the encryption is any good. Instead, the TLB must be organised with unit granularity. Further, all encrypted addresses generated in user mode are first remapped internally by the TLB to a pre-set range of logical addresses with the allocation serially ordered by 'first-come, first-served'. Since data that will later be accessed together tends also to be addressed for the first time in close sequence, this allows conventional cache lookahead policies to continue to operate successfully, on either logical or physical addresses.

Moreover, it has turned out to be possible to pass the unencrypted data address to the memory unit during the processing of load and store instructions, with no additional processing. We are nervous of the security implications, so we do not suggest that that should be done. However, the bare 32-bit address can be hashed or encrypted in a different way from there.

4. Performance

Table 1. Performance data, or1ksim test-suite instruction set add test.

@exit : cycles 315640, instructions 222006		
mode	user	super
register instructions	0.2%	0.2%
immediate instructions	7.3%	9.2%
load instructions	0.9%	2.8%
(cached)	(0.9%)	
store instructions	0.9%	0.0%
(cached)	(0.9%)	
branch instructions	1.0%	4.9%
jump instructions	1.1%	4.8%
no-op instructions	6.4%	15.8%
prefix instructions	11.5%	0.0%
move from/to SPR instructions	0.1%	2.7%
sys/trap instructions	0.5%	0.0%
wait states	24.7%	4.9%
(stalls)	(22.1%)	(3.8%)
(refills)	(2.7%)	(1.1%)
total	54.8%	45.2%

Branch Prediction Buffer			
hits	10328 (55%)	misses	8219 (44%)
right	8335 (44%)	right	6495 (35%)
wrong	1993 (10%)	wrong	1724 (9%)

User Data Cache			
read hits	2942 (99%)	misses	0 (0%)
write hits	2933 (99%)	misses	9 (0%)

The OpenRISC 'or1ksim' simulator (`opencores.org/or1k/Or1ksim`) has been modified heavily to run the prototype discussed here. It is now a cycle-accurate simulator, 800,000 lines of finished C code having been added to the original over two years through a sequence of prototypes. The first upgraded the 32-bit simulator to meet the 64-bit OpenRISC standard (`sf.net/p/or1ksim64ptb`), the second introduced the Rijndael encryption, the third pipelined the model, and so on. The code archive and development history is available at `sf.net/p/or1ksim64kpu` and analytics are at `nbd.it.uc3m.es/~ptb/or1ksim64KPU-stats`. The instruction function part comprises 30K lines, of 90K lines per model.

The processor instruction set tests from the or1ksim suite have been modified to run encrypted. The original tests ran in supervisor mode, which would not have tested the prototype because supervisor mode is unencrypted. Our modification (`sf.net/p/or1k64kpu-binutils`) of the OpenRISC port of the GNU 'gas' assembler v2.24.51 produced the encrypted machine code.

Table 1 displays the performance statistics from the modified instruction set add test ('is-add-test') of the or1ksim test-suite. The statically compiled executable contains 185628 machine code instructions, which occupy 742512 bytes in the 769454 byte executable, the rest being comprised of the executable file headers, symbol table, etc. Table 1 shows that when this test was run (successfully) to completion, 222006 instructions were executed, so there are few loops and subroutines (the code is largely built using assembler macros) in 315640 cycles. If one reckons with a 1GHz clock, then the speed was just over 700Kips (instructions per second).

In supervisor mode, pipeline occupation is just under 90%, at 892Kips for a 1GHz clock (wait states, cycles in which the pipeline fails to complete an instruction, comprise 4.9% of the 45.2% total), which one may take as a baseline for a single pipeline superscalar design. In user mode pipeline occupation is only 54.9%, as measured by numbers of non-wait states, for 549Kips with a 1GHz clock. Measured against supervisor mode, that is 61.6% of the unencrypted speed. Experiment shows that every extra codec stage drops throughput 2.5%.

The wait states are caused by unavoidable pipeline data hazards. Most (84%) are due to a load instruction feeding directly to an arithmetic instruction. A stall occurs because the data address for the load instruction is only calculated in execute stage, so the data cannot at that time already be available to the instruction sitting in read stage just behind.

5. Future work

The performance data indicates that a dual pipeline and on-the-fly instruction reordering would bring speed up over 70% of unencrypted running, and we plan to implement that. We will also model memory bus interactions more closely in order to optimise cache positioning and configuration.

An 'administrator' mode should be added between supervisor and user mode to run encrypted with privileges to support an encrypted operating system with a view to possible virtualisation.

A secure boot chain and secure operating system kernel and virtual machine in software can be offered as an alternative to the encrypted processor hardware – sensitive routines and data being confined to work in registers only while interrupts are masked.

6. Conclusion

A sophisticated superscalar pipeline design for a high-speed processor that works encrypted in user mode has been described here, with performance measured at 60-70% of unencrypted processing while embedding a 10-cycle (Rijndael) 64-bit encryption in this prototype. Registers, memory and buses contain encrypted user data, opaque to the operator and operating system, offering an avenue towards secure, remote high-speed computing in future.

References

1. P. T. Breuer, J. P. Bowen, A fully homomorphic crypto-processor design: Correctness of a secret computer, in: Proc. Intl. Symp. on Engineering Secure Software and Systems (ESSoS 2013), no. 7781 in Lecture Notes in Computer Science, Springer, Heidelberg, 2013, pp. 123–138. doi:10.1007/978-3-642-36563-8_9.
2. D. Molnar, T. Kohno, N. Sastry, D. Wagner, Tamper-evident, history-independent, subliminal-free data structures on prom storage -or- how to store ballots on a voting machine, in: Proc. 2006 IEEE Symposium on Security and Privacy, 2006, pp. 364–370. doi:10.1109/SP.2006.39.
3. E. Palomar, Z. Liu, J. P. Bowen, Y. Zhang, S. Maharjan, Component-based modelling for sustainable and scalable smart meter networks, in: Proc. IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2014), 2014, pp. 1–6. doi:10.1109/WoWMoM.2014.6918927.
4. P. T. Breuer, J. P. Bowen, Towards a working fully homomorphic crypto-processor: Practice and the secret computer, in: Proc. Intl. Symp. on Engineering Secure Software and Systems (ESSoS 2014), Vol. 8364 of Lecture Notes in Computer Science, Springer, Heidelberg, 2014, pp. 131–140. doi:10.1007/978-3-319-04897-0_9.
5. J. Daemen, V. Rijmen, The Design of Rijndael: AES – The Advanced Encryption Standard, Springer Verlag, 2002.
6. P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: Proc. EUROCRYPT'99, Advances in Cryptology, Springer, 1999, pp. 223–238.
7. O. Kömmerling, M. G. Kuhn, Design principles for tamper-resistant smartcard processors, in: Proc. USENIX Workshop on Smartcard Technology, USENIX Association, Berkeley, CA, 1999, pp. 9–20.
8. M. Hashimoto, K. Teramoto, T. Saito, K. Shirakawa, K. Fujimoto, Tamper resistant microprocessor, US Patent 2001/0018736 (2001).
9. F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, M. Russinovich, VC3: Trustworthy data analytics in the cloud using SGX, in: Proc. IEEE Symposium on Security and Privacy, 2015, pp. 38–54.
10. I. Anati, S. Gueron, S. P. Johnson, V. R. Scarlata, Innovative technology for CPU based attestation and sealing, in: Proc. 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, Intel, 2013.
11. P. T. Breuer, J. P. Bowen, Typed assembler for a RISC crypto-processor, in: Proc. Intl. Symp. on Engineering Secure Software and Systems (ESSoS 2012), no. 7159 in Lecture Notes in Computer Science, Springer, 2012, pp. 22–29. doi:10.1007/978-3-642-28166-2_3.
12. P. T. Breuer, J. P. Bowen, Certifying machine code safe from hardware aliasing: RISC is not necessarily risky, in: Software Engineering and Formal Methods, no. 8368 in Lecture Notes in Computer Science, Springer, Heidelberg, Heidelberg, 2014, pp. 371–388, proc. SEFM 2013 Collocated Workshops (OpenCert 2013). doi:10.1007/978-3-319-05032-4_27.
13. D. A. Patterson, Reduced instruction set computers, Communications of the ACM 28 (1) (1985) 8–21.