A Trojan Resistant Architecture

James Clay

State University of New York

jnclay@buffalo.edu

May 6, 2015

Overview

Summary

We present a hardware architecture similar to work by Beaumont et. al. [Beaumont et al., 2012] while utilizing a homomorphic cryptography based processor similar to [Breuer and Bowen, 2013].

Our design differs however in that we consider a cryptographic processor unit (KPU) in conjunction with a set of heterogeneous processors that use verification algorithms to ensure that the KPU executes properly.

Goals

Our work proceeds under the maliciously unreliable model.

- Assume that a trojan is present, can we hide the computations we're performing?
- Can we introduce enough fault tolerance to provide useful computation before failure?

Highlights

- Define homomorphic encryption
- Discuss components of a KPU
- Describe verification mechanisms
- Describe multiparty voting and SAFER PATH
- Discuss the architecture

Homomorphic Encryption

Homomorphic Encryption

An encryption scheme such that operations performed on encrypted data have the same effect as those acting on unencrypted data. For example, adding two encrypted numbers should result in the sum of their unencrypted counter parts.

Example Homomorphic Encryption

Example: Suppose we wish to add two n bit numbers, x and y. Let p, q denote two large (N^2 and N^5 respectively) random numbers. Let p be odd. We use these as private keys. Let m stand for a particular bit of the input. Let m' be a length N random number such that $m' = m \mod 2$. We encrypt each bit x_i and y_j as $c = m' + pq$, note that we select a random new q for each c. Thus an encrypted x and y would look like

$$c_x =< c_{x1}, c_{x2}, c_{x3}..., c_{xn} >$$

$$c_y =< c_{y1}, c_{y2}, c_{y3}..., c_{yn} >$$

Operations can then be performed "bitwise" (really on elements of the vector array that are $\approx N^6$ or N^7 bit integers).

Example Homomorphic Encryption

The AND gate: $c_{x \wedge y} =< c_{x1} \wedge c_{y1}, c_{x2} \wedge c_{y2}, c_{x3} \wedge c_{y3} ..., c_{xn} \wedge c_{yn} >$.

Here AND is the same as bitwise multiplication. Thus,

$$\begin{aligned} c_{xi} \wedge c_{yj} &= c_{xi} \times c_{yj} \\ &= (m'_{xi} + p(q_{xi})) \times (m'_{yi} + p(q_{yi})) \\ &= m'_{xi} m'_{yi} + p(q_{xi} q_{yi} + m_{xi} + m_{yi}) \end{aligned}$$

XOR can be defined similarly (e.g. add instead of multiply).

Example Homomorphic Encryption

Decryption from this state is relatively straightforward,

$$
\begin{aligned}
c_{xi} \wedge c_{yj} &= m'_{xi} m'_{yi} + p(q_{xi} q_{yi} + m_{xi} + m_{yi}) \\
\mathrm{DEC}(c_{xi} \wedge c_{yj}) &= (m'_{xi} m'_{yi} + p(q_{xi} q_{yi} + m_{xi} + m_{yi})) \mod p \mod 2 \\
&= (m'_{xi} m'_{yi}) \mod 2 \\
&= m'_{xi} \mod 2 \times m'_{yi} \mod 2 \\
&= m_{xi} m_{yi}
\end{aligned}
$$

Unfortunately...

For each bit in the original input you have a minimum of N^6 more bits!

You need to perform multiplication and randomizations for these N^6 more bits!

More issues...

Each operation on the encrypted text introduces "noise". [Gentry, 2009]'s contribution noted that one may re-encrypt without revealing the plaintext while reducing noise.

However, this re-encryption needs to be performed often and requires large computational overhead.

Improvements

Older software implementations would take roughly 30 minutes per bit operation.

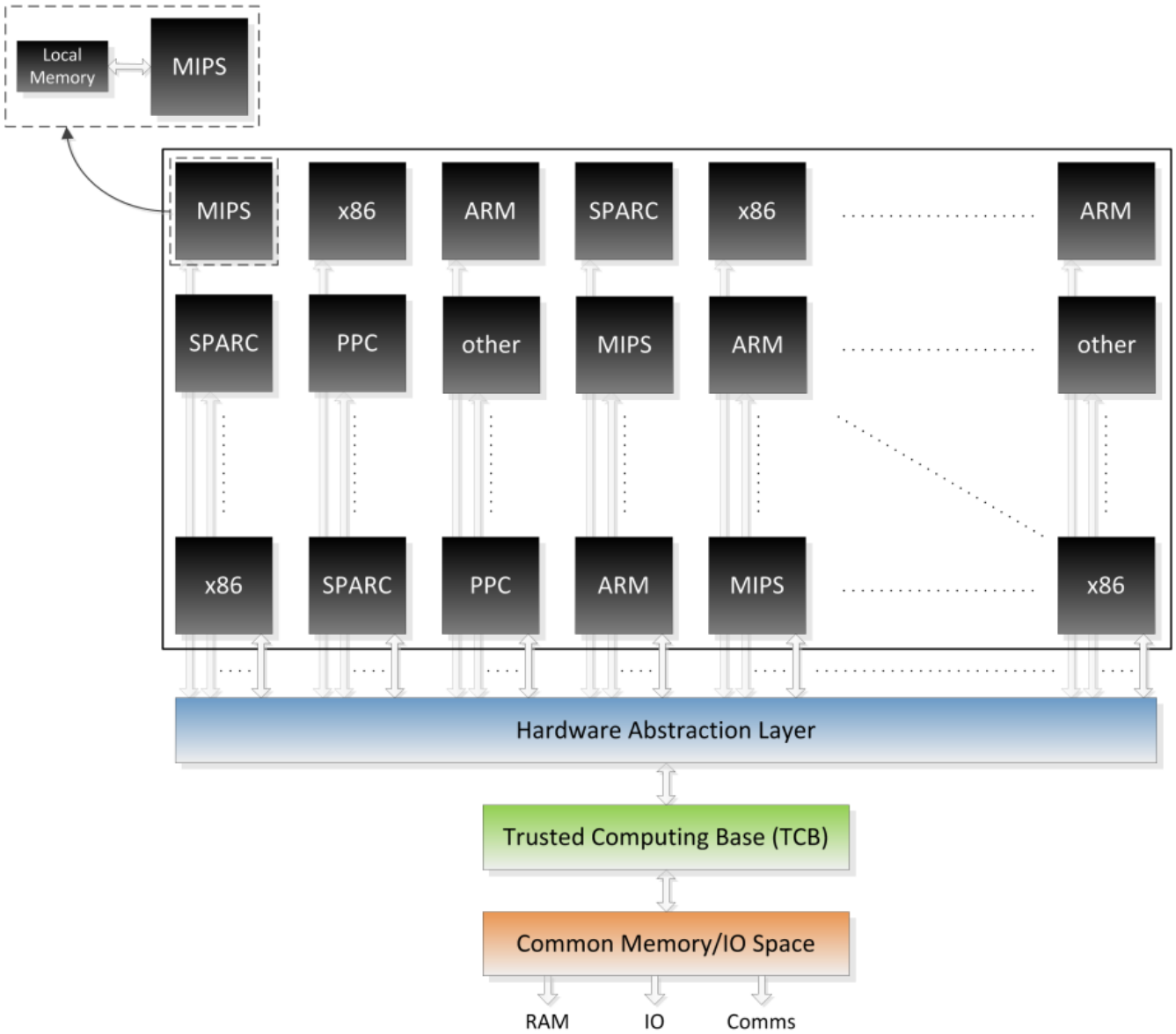
Newer implementations can perform single bit operations in about 1/2 a second [Ducas and Micciancio, 2015].

Related Work

[Breuer and Bowen, 2013] present a proof of correctness for a KPU and associated ALU. They do not provide layout or physical architecture or implementation detail.

[Beaumont et al., 2012] use a grid of heterogeneous processors to improve fault tolerance and quality of computation. Considers the rarity that all processors will fail or maliciously work in unison. Delineates execution based on instruction banks, each processor responsible for computing small pieces of a larger program.

SAFER-PATH


Local Memory
MIPS
MIPS
x86
ARM
SPARC
x86
ARM
SPARC
PPC
other
MIPS
ARM
other
x86
SPARC
PPC
ARM
MIPS
x86
Hardware Abstraction Layer
Trusted Computing Base (TCB)
Common Memory/IO Space
RAM
IO
Comms

Related work

Theoretical computer science uses the notion of verifying the results of computation efficiently. There has been significant work in software engineering towards this goal as well [Walfish and Blumberg,].

Performing efficient verification is crucial for our application: we want the CPU grid verifying the KPU workhorse. [Chung et al., 2010] give strong evidence that we can perform verification in log T time. Whether it fully applies needs to be investigated further.

Anatomy of a KPU

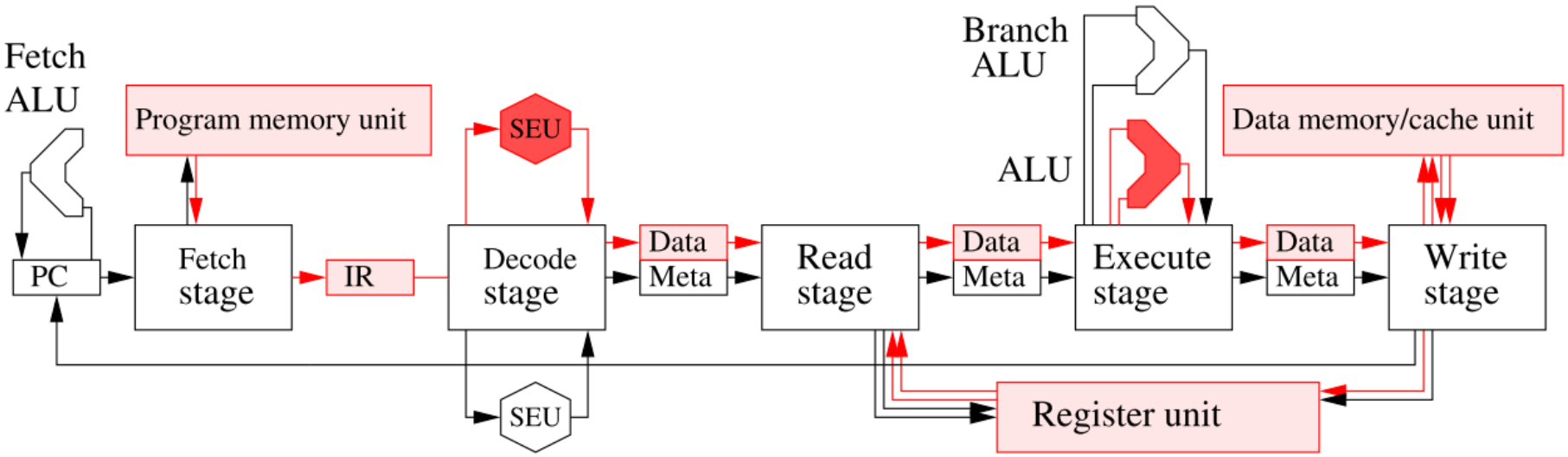


Figure: KPU: Red areas hold encrypted data [Breuer and Bowen, 2013].

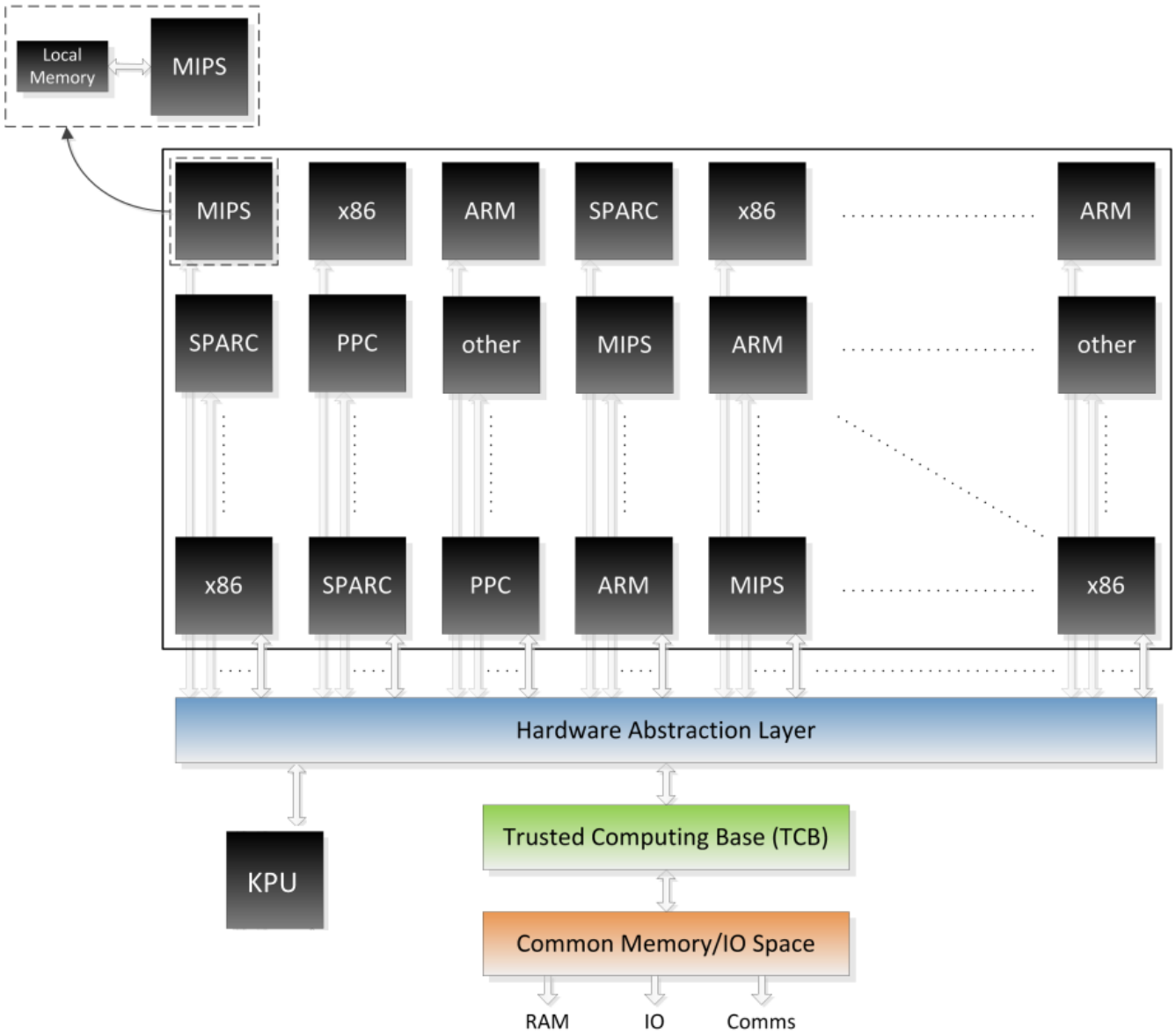
Note: addresses aren't encrypted. Might be addressable with CryptoPage [Duc and Keryell, 2006].

Verification

Basic verification scheme: we send one of $c_1, c_2, c_3, \mathcal{D}_e$ to each OTS (off-the-shelf) processor, where c_1, c_2 are inputs, c_3 is output, and $\mathcal{D}_e$ is the described bit operation.

Using a technique described in [Kaminski, 1989] and [Kozen, 1992] we perform integer multiplication verification in $O(n \log n \log \log n)$. We can do so with high probability in time linear with respect to n. An analysis could be performed to determine whether the probabilistic algorithm is sufficient for our purposes.

Architecture



Scheme

- Encrypted program and inputs fed in. Expect encrypted output.
- The KPU acts as a slow, but secure, processor. We verify its operational correctness across all of the other CPUs which are even slower at the same task.
- We use verification, rather than brute force to ensure that each task is performed correctly.
- Voting is then conducted, if a particular CPU votes against the majority we can flag as suspicious.
- Supposing KPU fails the processor farm can run the same algorithms albiet at a much slower pace providing some protection against kill switch like backdoors.

Benefits

- No information about inputs, outputs, or intermediary computations are leaked.
- Only vulnerable to side channel analysis, even with full visibility of the hardware functioning.
- Killswitch and trojan manipulation significantly hindered.

Drawbacks

- Currently slow.
- KPU needs to hide its address space in some way.
- Facilities KPU uses like random number generation need to be closely scrutinized

Further work

- There's already been work done in getting strongly pseudo random numbers from non-random sources [Kamara and Katz, 2008]. See if we can get a scheme whereby the hardware/software work in unison to provide a secure random number.
- Add something like CryptoPage to KPU [Duc and Keryell, 2006].
- Explore more efficient homomorphic mechanisms

References I

Beaumont, M., Hopkins, B., and Newby, T. (2012). Safer path: Security architecture using fragmented execution and replication for protection against trojaned hardware. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1000–1005. EDA Consortium.

Breuer, P. T. and Bowen, J. P. (2013). A fully homomorphic crypto-processor design. In *Engineering Secure Software and Systems*, pages 123–138. Springer.

Chung, K.-M., Kalai, Y., and Vadhan, S. (2010). Improved delegation of computation using fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2010*, pages 483–501. Springer.

Duc, G. and Keryell, R. (2006). Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 483–492. IEEE.

Ducas, L. and Micciancio, D. (2015). Fhew: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015*, pages 617–640. Springer.

References II

Gentry, C. (2009). *A fully homomorphic encryption scheme*. PhD thesis, Stanford University.

Kamara, S. and Katz, J. (2008). How to encrypt with a malicious random number generator. In *Fast Software Encryption*, pages 303–315. Springer.

Kaminski, M. (1989). A note on probabilistically verifying integer and polynomial products. *Journal of the ACM (JACM)*, 36(1):142–149.

Kozen, D. (1992). *The design and analysis of algorithms*. Springer Science & Business Media.

Walfish, M. and Blumberg, A. J. Verifying computations without reexecuting them: from theoretical possibility to near-practicality.

The End