A Fully Homomorphic Crypto-Processor Design
Correctness of a Secret Computer

Peter T. Breuer\textsuperscript{1,*} and Jonathan P. Bowen\textsuperscript{2,**}

\textsuperscript{1} Department of Computer Science, University of Birmingham, UK
ptb@cs.bham.ac.uk
\textsuperscript{2} Faculty of Business, London South Bank University, UK
jonathan.bowen@lsgu.ac.uk

Abstract. A KPU is a replacement for a standard CPU that natively runs encrypted machine code on encrypted data in registers and memory – a ‘crypto-processor unit’, in other words. Its computations are opaque to an observer with physical access to the processor but remain meaningful to the owner of the computation. In theory, a KPU can be run in simulation and remain as secure (or otherwise) as in hardware. Any block cipher with a block-size of about a word is compatible with this developing technology, the long-term aim of which is to make it safe to entrust data-oriented computation to a remote environment.

Hardware is arranged in a KPU to make the chosen cipher behave as a mathematical homomorphism with respect to computer arithmetic. We describe the architecture formally here and show that ‘type-safe’ programs run correctly when encrypted.

1 Introduction

A KPU is a replacement for a standard CPU (‘central processor unit’) that natively runs encrypted machine code on encrypted data in registers and memory. The term ‘KPU’ is derived from ‘crypto-processor’, and while the latter has been used for several hardware-based units aimed at helping overall system security (see, for example, \cite{3,10,15}), we mean it in the literal sense of a complete general purpose processor that has been architected to perform all its computations encrypted. Any block cipher is compatible provided that the block-size is not impractical – it dictates the physical size of an information word. The technology is aimed at allowing data-oriented applications such as fluid dynamics computations, image processing, even cryptography, to run in an insecure environment in relative security.

An observer can recognize control flow (jumps, branches, etc.), but the meaning of the data is hidden by the encryption. An observer may see the calculation

$$43 \# 43 = 21234089$$

but that it represents $1 + 1 = 2$ is known only to the owner of the computation.

\* Some work reported here was carried out as Visiting Fellow at LSBU.
\** Jonathan Bowen acknowledges the support of Museophile Limited.

Everything a programmer needs to know about a KPU is summarized in Box 1. The machine code is an encrypted form of the standard RISC instruction set. The interested reader will find an ‘on the metal’ instruction format in the patent [1], which sets out design rules that result in a correctly working KPU, whatever the block cipher chosen, plus a reference implementation.

This paper deals with correctness. As well as setting out the KPU design and design principles, we show that a KPU running an encrypted machine code program runs ‘correctly’: it generates machine states that are encryptions of the machine states expected in an ordinary RISC CPU running the corresponding unencrypted machine code program. The only proviso is that the running program does not ‘break the conventional apartheid between program and data’, which we characterize as type-safe for a KPU (also known as ‘crypto-safe’) below. That means in particular that a compiler for the KPU needs to be run outside of the KPU.

**Box 1. A KPU** is a CPU that natively processes mixed encrypted and unencrypted data in general purpose registers and memory. It executes encrypted RISC machine code instructions on:

- encrypted data and data addresses,
- unencrypted program addresses.

Instructions must be programmed with encryption type in mind. Arithmetic instructions apply to encrypted data, jumps to unencrypted program addresses. But...

- memory load/store instructions are polymorphic: they copy data whether encrypted or unencrypted.

**Definition 1.** A program is type-safe for a KPU (‘crypto-safe’) if those KPU machine instructions that work on encrypted data always get encrypted data on which to work during execution of the program, while those instructions that work on unencrypted data always get unencrypted data on which they can work.

The need for such a notion arises from the fact that a mix of encrypted and unencrypted data is always circulating inside a KPU and through memory and registers. While data and data addresses are encrypted, program addresses are not. More generally, program address encryption needs to be different from data and data address encryption, but we will suppose the program address encryption is null for the purposes of this exposition. On the one hand the distinction is physically mandated: the circuit that updates the program counter is distinct from the circuit that does the general arithmetic, so the encryptions may be different without interfering. On the other hand the two encryptions ought to be different for cryptographic reasons: the usual change in the program counter from cycle to cycle is an increment by 4 on a 32-bit machine, so valuable information could be garnered were the counter to be observed.

Type-safety in the KPU is explored in more detail in [2]. The special KPU RISC+CRYPT assembly is a typed language, and those assembly language programs that type-check correctly are shown in [2] to be type-safe for the KPU.

This paper is structured as follows: first, a top-down view of KPU design is given in Section 2 and 3, then we give a description of a RISC CPU in Section 4.