Typed Assembler for a RISC Crypto-Processor

Peter T. Breuer[1,⋆] and Jonathan P. Bowen[2,⋆⋆]

[1] Department of Computer Science, University of Birmingham, UK
ptb@cs.bham.ac.uk
[2] Faculty of Business, London South Bank University, UK / Museophile Limited, UK
jonathan.bowen@lsbu.ac.uk
http://www.jpbowen.com

Abstract. Our general purpose crypto-processor runs RISC machine code in an encrypted environment, reading encrypted inputs and generating encrypted outputs while maintaining data encrypted in memory. Its intended use is secure remote processing. However, program addresses are processed unencrypted, resulting in a mix of encrypted and unencrypted data in memory and registers at any time. An aspect of compiling for it is typing the assembler code to make sure that those instructions that expect encrypted data always get encrypted data at execution time, and those that expect unencrypted data get unencrypted data. A type inference system is specified here and transformed into an executable typing algorithm, such that a type-checked asembler program is guaranteed type-safe.

1 Introduction

The term 'crypto-processor' has been used to label several hardware-based solutions aimed at helping system security [2,5,7]. Our crypto-processor [1] is a general purpose unit that performs computations on mixed unencrypted and encrypted data held at encrypted addresses in memory. Its instruction set is standard RISC [6] but interpreted on encrypted data. In other words, when the processor computes 43+43=1234789 via an '**addiu**' machine instruction, it may well be computing an encrypted version of 1+1=2 but the latter 'translation' should be unknown to all but the remote owner. The processor characteristics are summarised in Box 1.

The intention is to be able to hide data and process from prying eyes in a remote computing environment – a cloud, for example. The hardware design is intended to make that feasible and secure even in simulation. The detail of the design is such that: (i) arithmetic and logical machine code instructions act on encrypted data and encrypted data addresses and produce

Box 1. A crypto-processor ...

... for the purposes of this article is a RISC CPU that manipulates mixed encrypted and unencrypted data in general purpose registers and memory. It runs on:

- encrypted data values and addresses, giving encrypted results where appropriate;
- unencrypted program addresses; while
- data and function codes embedded in the machine code are encrypted, register indices unencrypted.

⋆ This paper was initiated through an academic visit by the first author to London South Bank University as a Visiting Research Fellow during 2011–12.
⋆⋆ Jonathan Bowen is grateful for financial support from Museophile Limited.

G. Barthe, B. Livshits, and R. Scandariato (Eds.): ESSoS 2012, LNCS 7159, pp. 22–29, 2012.

encrypted data; (ii) control instructions (branches, jumps, etc) act on unencrypted program addresses; (iii) data embedded in machine code instructions and instruction function codes are encrypted; (iv) data or data addresses in memory and registers are encrypted; (v) program addresses in memory and registers are unencrypted; (vi) memory is divided into heap, which contains encrypted data, and stack, which contains mixed encrypted and unencrypted data.

As a consequence of the program addresses remaining unencrypted, programs are generally arranged in contiguous areas of memory. While that is certainly an advantage for cacheing, there is a different physical reason behind it: the circuit that updates the program counter is physically distinct from the circuitry that does the general arithmetic in the CPU. There is also a cryptographic reason: since the usual change in the program counter from cycle to cycle is a straightforward increment, plenty of information on the encryption could be gathered were the program counter to be observed (nevertheless, there is no fundamental design impediment to encrypting program addresses too).

That is a brief overview of how our crypto-processor works, but where is the necessity for type-checking assembler code? The answer is that at least one standard RISC machine instruction, the 'jump register' (**jr**) instruction, expects to read an unencrypted program address value from a register. And it is not the only instruction to expect unencrypted data – or to generate it. Thus, at any given moment, memory and registers in our crypto-processor contain a *mix* of encrypted and unencrypted data. That raises the question of whether a program for the crypto-processor is properly *type-safe*:

Definition 1. *A program is* type-safe *for a crypto-processor if those machine instructions in the program 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.*

This paper sets out a type-checking algorithm for assembler programs written for the crypto-processor, such that when a program type-checks, then it is type-safe.

The organisation of this paper is as follows: Section 2 introduces type-checking. Section 3 details the RISC+CRYPT assembler language (in an unencrypted representation) for our crypto-processor, and gives the inference rules of a type-system. Section 4 turns the inference system into an algorithm which deduces types. Section 5 elaborates the type-system (and associated type-checking algorithm) so that a RISC+CRYPT program which type-checks successfully is guaranteed type-safe.

2 Type-Checking

Successfully type-checking a program guarantees that:

(a) the distribution of encrypted and unencrypted data in the registers at every pass through the encrypted machine code always satisfies the same pattern at the same point in the code;
(b) the distribution of encrypted and unencrypted data in the registers is compatible with the instruction operating on them at every point.

The first claim says that the pattern of encrypted and unencrypted data in registers is stable as loops are repeated, subroutines are called, etc. The simple system of encrypted '**c**' and unencrypted '**u**' data types used by the analysis is shown in Box 2.

Box 2. A simple system of types ...

... describes register contents each processor cycle:

```
c   // encrypted data
u   // unencrypted data
x   // type variables match all data
```

For example, a pattern of types in registers 0 to 31 before and after the **move t2 t3** (i.e., '$\mathbf{t2} \leftarrow \mathbf{t3}$' for registers **t2**, **t3**) instruction, is shown in Box 3. After the instruction, the type of the data in register **t2** may be either encrypted or unencrypted, but it is constrained to be the same as the type of the data in register **t3**. The type signature is expressed as follows:

$$\mathbf{move\ t2\ t3} :: [\mathbf{t3} : \boldsymbol{x}] \rightarrow [\mathbf{t2} : \boldsymbol{x},\ \mathbf{t3} : \boldsymbol{x}]$$

Variable $\boldsymbol{x}$ matches any data type and every register not explicitly mentioned remains unchanged in type. The notation will be used throughout this article.

What of the second guarantee afforded by the typing algorithm? The claim is that the before-after patterns of register occupation around each instruction conform to the semantics of the instruction. In the case of the **move t2 t3** instruction, that means that whatever kind of data was in register 11 (**t3**) at the beginning, is also the kind of data found in register 10 (**t2**) afterwards, and nothing else has changed, just as in Box 3.

Box 3. A register type pattern ...

... around the **move t2 t3** instruction (i.e., $\mathbf{t2} \leftarrow \mathbf{t3}$):

reg.	0	1	2	3	...	10	11	...
before	$\boldsymbol{x}_0$	**c**	$\boldsymbol{x}_2$	**u**	...	**c**	$\boldsymbol{x}_{11}$	...
after	$\boldsymbol{x}_0$	**c**	$\boldsymbol{x}_2$	**u**	...	$\boldsymbol{x}_{11}$	$\boldsymbol{x}_{11}$	...

Registers **t2**, **t3** are registers 10, 11 respectively.

The crypto-processor assembler contains 'CRYPT' pseudo-instructions that the compiler translates to plain RISC machine code, but which are there to allow assembler typing to proceed with accuracy. Box 4 lists these succinctly. They deal with data transfers to and from the stack area, which are implemented using RISC add, load and store machine instructions. However, the machine code may access anywhere in memory, both stack and heap, and those two areas are treated very differently by our analysis: heap may only contain encrypted data in our design, while stack may contain both encrypted and unencrypted data (a polyvalent stack is necessary to the design because some RISC machine instructions – jumps and branches – require unencrypted program addresses in registers, which data needs to be saved on the stack during subroutine calls). The extra CRYPT pseudo-instructions allow the type analysis of the assembler to adequately distinguish the two areas of memory.

Box 4. The assembly language ...

... for the crypto-processor includes 'CRYPT' pseudo-instructions in addition to RISC assembler:

- **push** n, **pop** n – in-/decrease stack by n words;
- **pushu** r, **pushc** r – append to stack one new word copied from plaintext/encrypted register r;
- **popu** r, **popc** r – displace last word of plaintext/encrypted stack content to register r;
- **putu** $r\ n$, **putc** $r\ n$ – copy plaintext/encrypted contents of register r to n'th stack word;
- **getu** $r\ n$, **getc** $r\ n$ – copy the plaintext/encrypted n'th stack word to register r.

The RISC part of the assembler instruction set is listed in Box 5. It is entirely standard and translates directly to machine code instructions.

Box 5. RISC assembly language.

lui $r\ n$	// Set reg. content.
sb $r_1\ n(r_2)$	// Store byte to mem.
lb $r_1\ n(r_2)$	// Load byte from mem.
sw $r_1\ n(r_2)$	// Store word to mem.
lw $r_1\ n(r_2)$	// Load word from mem.
jr r	// Jump to addr. in reg.
j a	// Jump to addr.
jal a	// Jump and link.
bnez $r\ a$	// Branch if reg. $\neq 0$.
nop	// No-op, do nothing.
move $r_1\ r_2$	// Copy from reg. to reg.
ori $r_1\ r_2\ n$	// Arithmetic bitwise or.
addiu $r_1\ r_2\ n$	// Arithmetic add op.
...	// ...

Embedded data n is encrypted, embedded program addresses a and register indices r are unencrypted.

3 Assembler Typing

This section will set out a type system based on the **c**, **u** types for the crypto-processor assembler code.

Most machine instructions do not perturb the ordinary linear flow of control through a program. These *linear* instructions comprise all instructions apart from jumps and branches. When a linear instruction i_a at address a executes, control inevitably passes afterward to the *next* program instruction after it in positional sequence in memory. In a RISC 32-bit MIPS [4] machine, the next instruction is at address $a+4$ (4 bytes further on). To avoid prejudice we set:

Definition 2. *a' is the address of the* successor *instruction sited immediately beyond the instruction i_a at address a in the program code.*

We will use a' throughout this paper in place of any particular increment $a+\text{length}(i_a)$.

Type signatures for linear assembler instructions can be expressed in the notation of Sect. 2, as shown in Box 6. For example, the **lui** $r\ n$ instruction sets the the content of a register r to the encrypted value $2^{16}n$ (n is supplied as an encrypted value embedded in the instruction itself), and thus its type signature is given as $[\,] \to [r : \mathbf{c}]$ in Box 6.

Placing two instructions of signatures $t_1 \to t_2$ and $t_3 \to t_4$ in sequence is only possible if the types t_2 and t_3 can be reconciled. If type t_2 says the type of register 1 is **c** and type t_3 says it is **u**, then it is not possible. But reconciliation, if possible, yields:

Box 6. Linear RISC+CRYPT signatures.

lui $r\ n$	$:: [\,] \to [r : \mathbf{c}]$
sb $r_1\ n(r_2)$	$:: [r_1, r_2 : \mathbf{c}] \to [r_1, r_2 : \mathbf{c}]$
lb $r_1\ n(r_2)$	$:: [r_2 : \mathbf{c}] \to [r_1, r_2 : \mathbf{c}]$
sw $r_1\ n(r_2)$	$:: [r_1, r_2 : \mathbf{c}] \to [r_1, r_2 : \mathbf{c}]$
lw $r_1\ n(r_2)$	$:: [r_2 : \mathbf{c}] \to [r_1, r_2 : \mathbf{c}]$
nop	$:: [\,] \to [\,]$
move $r_1\ r_2$	$:: [r_2 : \boldsymbol{x}] \to [r_1, r_2 : \boldsymbol{x}]$
ori $r_1\ r_2\ n$	$:: [r_2 : \mathbf{c}] \to [r_1, r_2 : \mathbf{c}]$
addi $r_1\ r_2\ n$	$:: [r_2 : \mathbf{c}] \to [r_1, r_2 : \mathbf{c}]$
...	
putc $r\ n$	$:: [r, \mathbf{sp} : \mathbf{c}] \to [r, \mathbf{sp} : \mathbf{c}]$
putu $r\ n$	$:: [r : \mathbf{u}, \mathbf{sp} : \mathbf{c}] \to [r : \mathbf{u}, \mathbf{sp} : \mathbf{c}]$
getc $r\ n$	$:: [\mathbf{sp} : \mathbf{c}] \to [r, \mathbf{sp} : \mathbf{c}]$
getu $r\ n$	$:: [\mathbf{sp} : \mathbf{c}] \to [r : \mathbf{u}, \mathbf{sp} : \mathbf{c}]$
push n	$:: [\mathbf{sp} : \mathbf{c}] \to [\mathbf{sp} : \mathbf{c}]$
pop n	$:: [\mathbf{sp} : \mathbf{c}] \to [\mathbf{sp} : \mathbf{c}]$

Definition 3. Sequential composition:

$$t_1 \to t_2\ ;\ t_3 \to t_4 \triangleq \text{unify}(t_2, t_3)(t_1 \to t_4)$$

where 'unify' delivers the variable bindings required to reconcile pattern t_2 with t_3, and applies them to the type $t_1 \to t_4$, giving the type of the sequential composition.

Other program type calculations are more complicated. In general two type patterns t_1, t_2 (written '$t_1 \to t_2$') have to be developed for every instruction address a in a program: t_1 is the most general register type pattern that the instruction may validly encounter when it starts; t_2 is that which subsequently obtains at program termination (after the instruction at some address b). For a subroutine, exit is just after the **jr** that returns control to caller.

Box 7. Simple RISC+CRYPT typing rules ...

... in terms of the instruction $[i_a]$ at address a and the type at the next instruction address a' after a by position:

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [\,] \to [r{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{lui}\; r\; n]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r_1, r_2{:}\mathbf{c}] \to [r_1, r_2{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{sb}\; r_1\; n(r_2)]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r_2{:}\mathbf{c}] \to [r_1, r_2{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{lb}\; r_1\; n(r_2)]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r_1, r_2{:}\mathbf{c}] \to [r_1, r_2{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{sw}\; r_1\; n(r_2)]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r_2{:}\mathbf{c}] \to [r_1, r_2{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{lw}\; r_1\; n(r_2)]$$

$$\frac{}{T \vdash a :: [r{:}\mathbf{u}] \to [r{:}\mathbf{u}]}[\mathbf{jr}\; r]$$

$$\frac{T \vdash b :: t_1 \to t_2}{T \vdash a :: t_1 \to t_2}[\mathbf{j}\; b]$$

$$\frac{T \vdash b :: t_1 \to t_2 \qquad T \vdash a' :: t_3 \to t_4}{T \vdash a :: [\,] \to [\mathbf{ra}{:}\mathbf{u}] \;;\; t_1 \to t_2 \;;\; t_3 \to t_4}[\mathbf{jal}\; b]$$

$$\frac{T \vdash b[r : \mathbf{c}] :: t_1 \to t_2 \quad T \vdash a'[r : \mathbf{c}] :: t_1 \to t_2}{T \vdash a :: [r : \mathbf{c}] \to [r : \mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{bnez}\; r\; b]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: t_1 \to t_2}[\mathbf{nop}]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r_2{:}\boldsymbol{x}] \to [r_1, r_2{:}\boldsymbol{x}] \;;\; t_1 \to t_2}[\mathbf{move}\; r_1\; r_2]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r_2{:}\mathbf{c}] \to [r_1, r_2{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{ori}\; r_1\; r_2\; n]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r_2{:}\mathbf{c}] \to [r_1, r_2{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{addiu}\; r_1\; r_2\; n]$$

$$\ldots$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r, \mathbf{sp}{:}\mathbf{c}] \to [r, \mathbf{sp}{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{putc}\; r\; n]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [r{:}\mathbf{u}, \mathbf{sp}{:}\mathbf{c}] \to [r{:}\mathbf{u}, \mathbf{sp}{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{putu}\; r\; n]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [\mathbf{sp}{:}\mathbf{c}] \to [r, \mathbf{sp}{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{getc}\; r\; n]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [\mathbf{sp}{:}\mathbf{c}] \to [r{:}\mathbf{u}, \mathbf{sp}{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{getu}\; r\; n]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [\mathbf{sp}{:}\mathbf{c}] \to [\mathbf{sp}{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{push}\; n]$$

$$\frac{T \vdash a' :: t_1 \to t_2}{T \vdash a :: [\mathbf{sp}{:}\mathbf{c}] \to [\mathbf{sp}{:}\mathbf{c}] \;;\; t_1 \to t_2}[\mathbf{pop}\; n]$$

Definition 4. *A collection of types $t_1 \to t_2$ indexed by entry addresses a is called a* theory. *We write*

$$T \vdash a :: t_1 \to t_2$$

for 'theory T lists the type $t_1 \to t_2$ against address a'.

In an actual program run, the register type pattern encountered by any particular instruction i_a at address a may be strictly less general than the type t_1 recorded in theory T. It will be the result $\sigma(t_1)$ of a substitution σ for type variables in t_1. The register type pattern at the end of the run will then match $\sigma(t_2)$.

The deduction rules of a type theory T for our cryptoprocessor are given in Box 7. Each rule is associated with a single program address a, and the instruction i_a located at that address. For the branch rule, both possible continuations after the branch test must give rise to the same register type pattern at program exit. The branch test requires an encrypted datum in register r and the following notation helps express the rule succinctly:

Definition 5. $T \vdash b[r : t] :: t_1 \to t_2 \;\triangleq\; T \vdash b :: t_3 \to t_4$
where $t_3 \to t_4$ is such that $[r : t] \to [r : t] \;;\; t_1 \to t_2 = [r : t] \to [r : t] \;;\; t_3 \to t_4$.

(two branch types become equal after substituting t for the type of r on entry to both).

4 The Basic Algorithm

Box 8. Altered rules ...

... for calculating the fixpoint type theory $T_n = T_{n-1}$ of a program, in terms of the instruction $[i_a]$ at address a and the following instruction address a'. All other rules from Box 7 have T_n substituted for T throughout.

$$\frac{}{T_0 \vdash a :: [0{:}\boldsymbol{x}_{a0}, 1{:}\boldsymbol{x}_{a1}, \dots] \to [0{:}\boldsymbol{y}_{a0}, 1{:}\boldsymbol{y}_{a1}, \dots]}\,[*]$$

$$\frac{T_{n-1} \vdash b :: t_1 \to t_2}{T_n \vdash a :: t_1 \to t_2}\,[\mathbf{j}\ b,\ b \le a]$$

$$\frac{T_{n-1} \vdash b[r : \mathbf{c}] :: t_1 \to t_2 \quad T_n \vdash a'[r{:}\mathbf{c}] :: t_1 \to t_2}{T_n \vdash a :: [r{:}\mathbf{c}] \to [r{:}\mathbf{c}]\ ;\ t_1 \to t_2}\,[\mathbf{bnez}\ r\ b,\ b \le a]$$

$$\frac{T_{n-1} \vdash b :: t_1 \to t_2 \qquad T_n \vdash a' :: t_3 \to t_4}{T_n \vdash a :: [\,] \to [\mathbf{ra} : \mathbf{u}]\ ;\ t_1 \to t_2\ ;\ t_3 \to t_4}\,[\mathbf{jal}\ b]$$

To that end, define T_n as the n'th iteration of a series leading to the final theory T that is a fixpoint of the iteration. Initially, the theory assigns to address a the 'any' pattern, in which all registers are bound to different type variables and inputs are not related to outputs (rule [*] in Box 8). A theory T_{n-1} in the sequence is used to help construct the next theory T_n, $n > 0$. by substitution for free variables in the types at each address a, as shown in Box 8. In all other cases, the rules are just as given in Box 7 but with T replaced with T_n throughout.

When no improvement is obtained from T_{n-1} to T_n at any address a, then the fixpoint theory T has been reached. Substituting T for T_n and T_{n-1} in Box 8 shows that the fixpoint T satisfies the rules of Box 7. There are only a finite number of proper substitutions possible as steps of the algorithm, so the iteration does terminate. In practice the number of iterations required is approximately the number of backward jumps and branches plus subroutine calls in the code.

5 Taking Account of the Stack

Although we have supplied a type system, it is not the case that, as is, the system constrains a type-checked program to be type-safe. The problem is evident in the fragment:

```
putu t1 0; getc t1 0
```

which writes an unencrypted value to the stack and then recovers the same datum as an encrypted value. The type of register **t1** changes from **u** to **c** yet the content of the register does not change. The **getc** instruction encounters an unencrypted value on the stack where it expects an encrypted value, yet the fragment type-checks. 'Typeable' via the system given so far means that code is type-safe for the crypto-processor only under the hypothesis that the stack operations in the code are independently type-safe.

To remove that additional assumption, the types of the values in different *stack slots* have to be tracked. The size of the stack will from now on be denoted by an annotation on the right side of a list of register types $[r_1 : t_1, r_2 : t_2, \dots]^d_k$. The subscript k indicates the list is $k \ge 32$ long, and the last $k - 32$ list entries represent the stack slots, while the first 32 represent the registers proper. The size $d \le k - 32$ of the current stack frame

is indicated by the superscript on the list. Writing to the n'th from the bottom word on the stack in the current frame with **putu** $r\ n$ accesses the n'th of the last d list entries, which is entry number $k-d+n$ in the list.

The type rules of Box 7 are altered as shown in Box 9. In particular, the evidently too-loose typing given for **putu**, **getu** in Box 7 is mended here so that **getu** requires to act on a stack slot of type **u**, and **putu** creates a stack slot of type **u**.

Box 9. Extended type rules ...

... which track types through the stack. The other rules of Box 7 uniformly have $\restriction_k^d$ added to the type assignment lists, indicating that the list is of length k and the last d entries represent the current stack frame (the first 32 of k are the registers), and is unchanged through the rule. The rules are for instruction $[i_a]$ at address a in terms of the type at the following program address a'.

$$\frac{T\vdash a'::t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}{T\vdash a::[r{:}\mathbf{u}]_k^d\to[r{:}\mathbf{u},k{-}d{+}n{:}\mathbf{u}]_k^d\ ;\ t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}[\mathbf{putu}\,r\,n]$$

$$\frac{T\vdash a'::t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}{T\vdash a::[r{:}\mathbf{c}]_k^d\to[r{:}\mathbf{c},k{-}d{+}n{:}\mathbf{c}]_k^d\ ;\ t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}[\mathbf{putc}\,r\,n]$$

$$\frac{T\vdash a'::t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}{T\vdash a::[k{-}d{+}n{:}\mathbf{u}]_k^d\to[r{:}\mathbf{u},k{-}d{+}n{:}\mathbf{u}]_k^d\ ;\ t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}[\mathbf{getu}\,r\,n]$$

$$\frac{T\vdash a'::t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}{T\vdash a::[k{-}d{+}n{:}\mathbf{c}]_k^d\to[r{:}\mathbf{c},k{-}d{+}n{:}\mathbf{c}]_k^d\ ;\ t_1\restriction_k^d\to t_2\restriction_{k'}^{d'}}[\mathbf{getc}\,r\,n]$$

$$\frac{T\vdash a'::t_1\restriction_{k+n}^{d+n}\to t_2\restriction_{k'}^{d'}}{T\vdash a::[\,]_k^d\to[k{:}x_k\ldots k{+}n{-}1{:}x_{k+n-1}]_{k+n}^{d+n}\ ;\ t_1\restriction_{k+n}^{d+n}\to t_2\restriction_{k'}^{d'}}[\mathbf{push}\,n]$$

$$\frac{T\vdash a'::t_1\restriction_{k-n}^{d-n}\to t_2\restriction_{k'}^{d'}}{T\vdash a::[k{-}n{:}x_{k-n}\ldots k{-}1{:}x_{k-1}]_k^d\to[\,]_{k-n}^{d-n}\ ;\ t_1\restriction_{k-n}^{d-n}\to t_2\restriction_{k'}^{d'}}[\mathbf{pop}\,n]$$

$$\frac{T\vdash b::t_1\restriction_{32,d}^{d}\to t_2\restriction_{32,d}^{d}\qquad T\vdash a'::t_3\restriction_k^d\to t_4\restriction_{k'}^{d'}}{T\vdash a::[\,]_k^d\to[\mathbf{ra}{:}\mathbf{u}]_k^d\ ;\ t_1\restriction_k^d\to t_2\restriction_k^d\ ;\ t_3\restriction_k^d\to t_4\restriction_{k'}^{d'}}[\mathbf{jal}\,b]$$

All other rules of Box 7 uniformly have $\restriction_k^d$ added to the type lists, indicating that the type list is of length k and the last d entries represent the current stack frame (the first 32 of k are the registers) and the list length/stack size is unchanged by the rule. The 'subroutine return' instruction **jr** in particular does *not* modify the stack – it is the **jal** 'subroutine call' rule that does the job. It drops consideration of all callee frames for the parent.

Proposition 1. *In the type system of Box 9 for the crypto-processor of Sect. 1,* type-checked implies type-safe *for RISC+CRYPT assembler programs.*

Proof. (Sketch) 'Notice' that the type rules and the algorithm that computes types from them together define an abstract interpretation [3] of the program: the values obtained in a program run match the type patterns computed, if the typing algorithm succeeds. Then the values obtained match the typing rule corresponding to each instruction in the program, since the algorithm works by refining each rule at each site where it is applicable and only one rule is applicable at each program address, that corresponding to the instruction located there. If a set of values obtained during execution does not match the instruction's input expectations, then – since it does match the input of the corresponding typing rule – the typing rule concerned permits inputs that do not match the instruction's expectations. But each typing rule can be seen by inspection not to allow inputs that are outside the corresponding instruction's expected range. That proves the result by contradiction. □

Where this argument falls down for the case of the basic type system of Sect. 3 and Box 7 is that its rules for **getc**, **getu** do permit inputs on the stack that do not match the expected range – an encrypted value is expected in the referenced stack slot for **getc**, but an unencrypted value is allowed by that type system, for example. The system of Box 9 mends that defect by tracking types through the stack.

6 Conclusion

We have introduced a RISC 'crypto-processor' that processes data kept in encrypted form in memory and registers, along with a type-checking algorithm for its assembly language. A type-checked program is type-safe: at run-time, encrypted data is always encountered by every instruction that expects encrypted data, and unencrypted data is always encountered by every instruction that expects unencrypted data.

7 Future Work

It turns out that it is possible to type-check RISC machine code directly by adapting the type system here from assembler to machine code. That enables a pre-existing machine code program to be type-checked, encrypted instruction by instruction, and run safely on our crypto-processor in a potentially hostile environment, the encrypted results of the computation being returned securely to the remote owner. The patent [1] contends that the processor design is secure even in simulation, making its (virtual) export feasible.

References

1. Breuer, P.T.: Encrypted data processing, patent pending, UK Patent Office GB1120531.7 (November 2011)
2. Buchty, R., Heintze, N., Oliva, D.: Cryptonite – A Programmable Crypto Processor Architecture for High-bandwidth Applications. In: Müller-Schloer, C., Ungerer, T., Bauer, B. (eds.) ARCS 2004. LNCS, vol. 2981, pp. 184–198. Springer, Heidelberg (2004)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM Symposium on the Principles of Programming Languages, pp. 238–252. ACM (1977)
4. Hennessy, J.L.: VLSI processor architecture. IEEE Trans. on Computers 33(C), 1221–1246 (1984)
5. Oliva, D., Buchty, R., Heintze, N.: AES and the cryptonite crypto processor. In: Proc. CASES 2003: International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM (2003)
6. Patterson, D.A.: Reduced instruction set computers. Communications of the ACM 28(1), 8–21 (1985)
7. Sun, M.C., Su, C.P., Huang, C.T., Wu, C.W.: Design of a scalable RSA and ECC crypto-processor. In: Proc. ASP-DAC 2003: Asia and South Pacific Design Automation Conference. ACM (2003)