STALlion: A Simple Typed Assembly Language for Static Analysis *

Martín Nordio,[1]**, Francisco Bavera**[1] **Ricardo Medel,**[1,3,†]
Jorge Aguirre[1]**, Gabriel Baum**[1,2]

(1) Universidad Nacional de Río Cuarto, Departamento de Computación
Río Cuarto, Argentina
{nordio,pancho,jaguirre}@dc.exa.unrc.edu.ar

(2) Universidad Nacional de La Plata, LIFIA
La Plata, Argentina
gbaum@sol.info.unlp.edu.ar

(3) Stevens Institute of Technology,
New Jersey, EE.UU.,
rmedel@cs.stevens-tech.edu

2004

Abstract

Typed assembly languages have the goal of providing security guarantees, for example, for the limited use of resources in a host machine or the detection of autoupdate code. This work presents a simple typed assembly language which allows us to perform various kinds of static analysis tasks with the purpose of detecting flaws in the code security. The security policy we use guarantees type and memory safety. Moreover, wa can ensure that non-initialized variables are not read, and that there is no out-of-bound array accesses. The language we present, called STALlion, was designed in order to interpret a particular kind of imperative programs, more specifically abstract syntax tree.

Keywords: Mobile Code, Proof-Carrying Code, Security Properties, Programming Languages.

*This work was supported through grants from the SECyT-UNRC, the Agencia Córdoba Ciencia, and the NSF.
†This author's work was supported by the NSF project *CAREER: A Formally Verified Environment for the Production of Secure Software*–#0093362.

1 Introduction

Typed assembly languages are assembly languages that preserve the types of the registers. They have the goal of providing security guarantees, for example, for the limited use of resources in a host machine or the detection of autoupdate code. This kind of assemblies have received increasing attention since the invention of Proof-Carrying Code (PCC). PCC is a technique developed by Necula and Lee [9] whose aim is guaranteeing the safety of untrusted mobile code. Since its conception in 1998, this technique generated several active lines of research [1, 2, 3, 6, 8, 9, 11, 12, 15, 17], but there are still some open problems.

With the goal of resolving some open problems of PCC, we developed the framework Proof-Carrying Code based on Static Analysis (PCC-SA) [13], that guarantees the safe execution of mobile code based on PCC and static analysis techniques. This framework uses a high-level intermediate language in order to verify the code security. Moreover, we implemented a prototype to study the applicability of our approach in practice. This framework uses an *abstract syntax tree* (AST) with type annotations. These intermediate representations enable us to use static analysis techniques in order to generate and verify this type information and also to apply several code optimization techniques. The AST is then used by the code consumer, both to prove that the code complies with the security properties, and to generate object code.

AST resulted to be a suitable vehicle for performing some analysis on the code, since it allows for linear-time evaluation (as opposed to the typical exponential-time complexity of standard PCC techniques). This is achieved at the expense of overloading the code consumer with the job of generating the object code before executing it. One way to solve this problem is by means of the use of a typed assembly language, as well as the AST. The AST represents only the proof that the code verifies the consumer's security policies and does not contain the executable code. In this way, the code consumer only verifies that the AST satisfy the security policy and it "matches" the assembly.

Another important feature of a typed assembly is that it can be used to make security checkers. If we combine the assembly language with the AST we can obtain a hybrid system where certain proofs are made on the AST and other proofs can be made by a type checker on the assembly. Some proofs are made on the AST because either they cannot be made on the assembly or they are more efficient on the AST.

With the aim of implementing the above described idea, we designed a typed assembly language named STALlion. In contrast with other proposed assembly languages, STALlion interprets ASTs, and allows for various static analysis tasks. Furthermore, we can obtain an abstract syntax tree from a program written in the STALlion typed assembly language efficiently. Due to the relation between both, the algorithm which verifies that a program written in STALlion matches an AST is simple and efficient.

This paper is structured as follows: in section 2 we present the STALlion assembly language. We describe the abstract syntax tree in section 3. The compilation function from the AST to a STALlion program is presented in section 4. The related work are analysed in section 5. The last section is devoted to the conclusions of our work and some proposals of future work.

2 The STALlion Typed Assembly Language

A program P consists of a sequence $l : I$ of labels and instructions. Each register of the language has a certain type. The types of registers are *int*, *arrayint* and [*int*]. The *int* type represents integer type, the *arrayint* represents integer array type and [*int*] pointer to an integer. Arrays are formed by an integer which it represents bound array and the corresponding set of integers. For example, if we have an array of five elements, this is formed by the constant 5 (which it cannot be updated by the programmer) and five elements of *int* type (figure 2). The integer pointers are composed of the memory address which contains the value and the account of registers (of type integer pointer) that points to this memory address. This last information is used to make different kinds of alias analysis.

The language types are defined by the following grammar:

Types T	::=	int^w | $array_int^w$ | $[int]^w$
w	::=	$+$ | $-$

Figure 1: Types of STALlion

The types are tagged with an initialization flag, z which can be either $+$ or $-$ indicating an initialized or an unitialized value, respectively. We say that the *array_int* type is initialized ($array_int^+$) if all its constituent elements are initialized, otherwise it is unitialized ($array_int^-$).

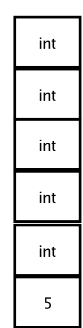


Figure 2: Example of a lenght-five array structure.

Program	P	::=	$<>$ | L: I ; P
Registers	R	::=	r_i, where $i \in N$
Labels	L	::=	l_i, where $i \in N$
Instructions	I	::=	B | J | E
BasicInstruc	B	::=	Mov $r_1 \leftarrow r_2$ | Mov $r_1 \leftarrow n$ where $n \in N$
		|	Add $r_1 \leftarrow r_2 + r_3$ | Sub $r_1 \leftarrow r_2 - r_3$
		|	Mul $r_1 \leftarrow r_2 * r_3$ | Div $r_1 \leftarrow r_2 / r_3$
		|	Comp $r_1 \leftarrow r_2 \quad r_3$
		|	bez_if r l_1 l_2 | bnz_while r l_1
		|	LoadArray $r \leftarrow r_2[r_3]$ | StoreArray $r[r_2] \leftarrow r_3$
		|	MemoryRead $r \leftarrow M[r_2]$ | MemoryWrite $M[r_2] \leftarrow r$
JumpInstruc	J	::=	Jump_up l | Jump_down l
ExtendedInstruc	E	::=	Call l | Ret l
		|	LoadPointer $r \leftarrow r_2$ | StorePointer $r \leftarrow r_2$
		|	Salloc r | Sfree r

Figure 3: The STALlion Typed Assembly Language

The language has a set of basic instructions, two unconditional jump instructions and a set of extended instructions. The basic instructions are *Mov, Add, Sub, Score, Times, bez_if, bez_while, LoadArray, StroreArray, MemoryRead, MemoryWrite y StackPointer.*

The *Mov* instruction moves the contents of a r_2 register to r_1 register. Given two registers r_2 and r_3, the *Add, Sub, Div, Mul* instructions add, subtract, multiply and divide the values stored in r_2 and r_3 respectively and store the result in the r_1 register. Given a register r, the *bez_if* instruction jumps to either the label l_1, if r is equal to 0, or, label l_2, otherwise. The *Comp* instruction compares the r_2 and r_3 registers and if r_2 is smaller than r_3 store the zero constant in r_1, otherwise, store the constan 1. Given a register r, the *bez_while* instruction jumps to the label l_1, if r is equal to 0. Although the *bez_if* instruction can be replace by the use of *bnz_while* instruction, we use both to easily and efficiently determine the presence of a loop or the presence

of a conditional sentence. The *LoadArray, StroreArray* instructions allow us to load the contents of a position of an array to a register and to store the contents of a register to an array, respectively. Furthermore, *MemoryRead* allows us to read from a memory position and to store its value to an r register and *MemoryWrite* allows us to store the contents of a r register in a memory position. Finally, the extended instructions allow us both to use pointers and to invoke functions.

2.1 Typing Rules

Own typing rules define a judgement $\Gamma \vdash P$ meaning that P is a well-typed assembly program in context Γ. The STALlion typing rules are defined as follows:.

$$\frac{\Gamma\ \vdash P \qquad n \in N}{\Gamma,\ r_j : int^+\ \vdash Mov\ r_j \leftarrow n;\ P}\ \textbf{mov_n_inst} \qquad \frac{\Gamma,\ r_i : int^+\ \vdash P}{\Gamma,\ r_j : int^+\ \vdash Mov\ r_j \leftarrow r_i;\ P}\ \textbf{mov_inst}$$

$$\frac{\Gamma,\ r_i : int^+,\ r_j : int^+\ \vdash P}{\Gamma,\ r_k : int^+\ \vdash Add\ r_k \leftarrow r_i\ +\ r_j;\ P}\ \textbf{add_inst} \qquad \frac{\Gamma,\ r_i : int^+,\ r_j : int^+\ \vdash P}{\Gamma,\ r_k : int^+\ \vdash Sub\ r_k \leftarrow r_i\ -\ r_j;\ P}\ \textbf{sub_inst}$$

$$\frac{\Gamma,\ r_i : int^+,\ r_j : int^+\ \vdash P}{\Gamma,\ r_k : int^+\ \vdash Mul\ r_k \leftarrow r_i\ *\ r_j;\ P}\ \textbf{mul_inst} \qquad \frac{\Gamma,\ r_i : int^+,\ r_j : int^+\ \vdash P}{\Gamma,\ r_k : int^+\ \vdash Div\ r_k \leftarrow r_i\ /\ r_j;\ P}\ \textbf{div_inst}$$

$$\frac{\Gamma,\ r_i : int^+,\ r_j : int^+\ \vdash P}{\Gamma,\ r_k : int^+\ \vdash Comp\quad r_k\quad r_i\ \ r_j;\ P}\ \textbf{comp_inst}$$

$$\frac{\Gamma,\ r_i : array_int^+,\ r_j : int^+\ \vdash P \qquad r_j < r_i.lenght}{\Gamma,\ r_l : int^+\ \vdash LoadArray\ r_l \leftarrow r_i[r_j];\ P}\ \textbf{loadArray_inst}$$

$$\frac{\Gamma,\ r_i : array_int^+,\ r_j : int^+\ \vdash P \qquad r_j < r_i.lenght}{\Gamma,\ r_l : int^+\ \vdash StoreArray\ r_i[r_j] \leftarrow r_l;\ P}\ \textbf{storeArray_inst}$$

Figure 4: Basic Rules of The Type System.

The **LoadArray, StoreArray** rules impose the property of constraint that non-existent array positions are not read and not written, respectivelly. We want to remind that the *array_int* type is made up by two fields: its lenght and $r_i's$ elements. If r_i is of *array_int* type, then $r_i.lenght$ contains its lenght. Therefore, $LoadArray\ r_l \leftarrow r_i[r_j]$ instruction is safe if r_j is smaller that the lenght of the array.

$$\frac{\Gamma\ \vdash P; l_1 : I_1 \qquad \Gamma \vdash l_2 : I_2 \qquad l_1, l_2 \in \Sigma(Labels) \qquad f_labels(l_1) = f_labels(l2) + 1}{\Gamma \vdash P; l_1 : I_1; l_2; I_2}\ \textbf{r_labels}$$

$$\frac{\Gamma\ \vdash P \qquad l_1, l_2 \in \Sigma(Labels) \qquad f_labels(l_1) > f_labels(l_2)}{\Gamma \vdash l_1 : Jump_up\ l_2;\ P}\ \textbf{jump_up_inst}$$

$$\frac{\Gamma\ \vdash P \qquad l_1, l_2 \in \Sigma(Labels) \qquad f_labels(l_2) > f_labels(l_1)}{\Gamma \vdash l_1 : Jump_down\ l_2;\ P}\ \textbf{jump_down_inst}$$

$$\frac{\Gamma\ \vdash P \qquad l_1, l_2, l_3 \in \Sigma(Labels) \qquad f_labels(l_1) < f_labels(l_2) < f_labels(l_3)}{\Gamma \vdash l_1 : bez_if\quad r\quad l_1\quad l_2;\ P}\ \textbf{bez_if_inst}$$

$$\frac{\Gamma\ \vdash P \qquad l_1, l_2 \in \Sigma(Labels) \qquad f_labels(l_1) < f_labels(l_2)}{\Gamma \vdash l_1 : bnz_while\quad r\quad l_1;\ P}\ \textbf{bnz_while_inst}$$

Figure 5: Labels Rules of the Type System.

Let *f_labels* be a function that given a label returns a natural number that represents the label number. Let $\Sigma(Labels)$ be the set of all labels of a STALlion program. The **r_labels** rule

determines that a program is well-typed, which among other things, ensures that label numbers are increasingly ordened.

$$\frac{\Gamma \vdash P \qquad r_2 : [int]^+}{\Gamma \vdash LoadPointer\ r \leftarrow r_2} \ \textbf{load_pointer_inst}$$

$$\frac{\Gamma \vdash P \qquad r : [int]^+ \qquad r_2 : int^+}{\Gamma \vdash StorePointer\ r \leftarrow r_2} \ \textbf{store_pointer_inst}$$

$$\frac{\Gamma \vdash P \qquad r : [int]^+ \qquad r.num_pointer = 0}{\Gamma \vdash Sfree\ r} \ \textbf{free_inst}$$

Figure 6: Pointer Rules of the Type System.

The **load_pointer_inst** rule determines when is safe to load a pointer r_2 in the r register. The associated semantics of **load_pointer_inst** is: $r_2.nun_pointers = r_2.num_pointers + 1$ and $r.nun_pointers = r.num_pointers - 1$. In words, $r_2.num_pointers$ contains the information of how pointers are pointed to this memory address. Then , in order to perform the $LoadPointer$ operation, the account of pointers pointing to the address held in r_2 is increased by 1, and its account pointed by r_2 is decreased by 1, since r_2 has changed its contained value. The **free_inst** rule determines that is safe to free a memory address if there is no pointer pointing to it (r.num_pointer=0).

3 The Abstract Syntax Tree

The *abstract syntax tree* (AST) is an abstract representation of a subset of C called Mini. This representation enables us to apply several static analysis, such as control flow and data flow analysis. Also, it can be used to apply code optimizations.

The prototype's abstract syntax trees are similar to a traditional AST, but the former include code annotations. These annotations show the status of the program objects, and they contain information about variable initializations, loop invariants, and variable ranges.

Each sentence of a program is represented by an AST. The nodes in an AST contain a label, information or references to the sub-sentences that compose the sentence, and a reference to the next sentence.

Each expression is represented by a graph. Two different labels are used when an array is accessed: `unsafe` and `safe`. These labels mean that it is not safe to access to such element of that array and that it is safe to access, respectively. By modifying the node label we avoid the necessity of include run-time checks.

The figure 7 presents the AST of the following example writted on Mini:

```
int ArraySum ( int index(0,0) ) {

    int [10] data;           /* Define an array */
    int value=1;             /* Define an initialization variable */
    int sum=0;               /* Define the summatory variable */

    while (index<10) {       /* Initialize the array */
        data[index]=value;
        value=value+1;
        index=index+1;
    }
    while (index>0) {        /* Calculate the summatory */
        sum=sum+data[index-1];
        index=index-1;
    }
```

```
        return sum;
 }
```

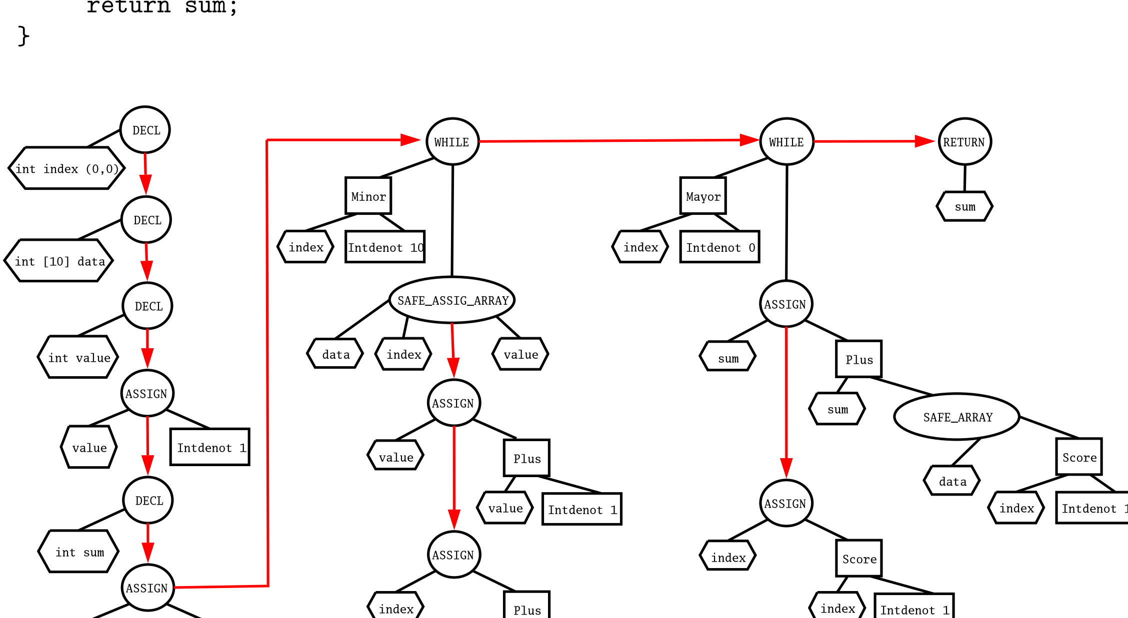


Figure 7: AST of the program example.

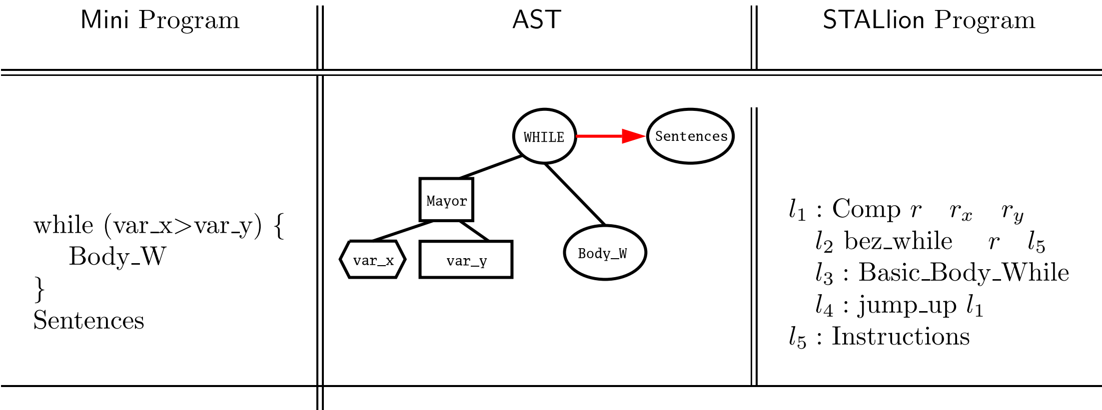
The circles in Figure 7 represent sentences, while the hexagons represent variables and the rectangles represent expressions. Arrows show the control flow, and straight lines join sentences with their attributes. The label DECL is used for declarations, ASSIGN for assign sentences, UNSAFE_ASSIG_ARRAY for array assign sentences, WHILE for loops, and RETURN for function return sentences. For example, note that the AST of the first loop includes the logic condition ($index < 10$) and the body of the loop. The AST of the body includes three assign sentences, the first of them assigns the value *value* to the element *index* of the array *data*. So, it is labeled UNSAFE_ASSIG_ARRAY.

4 Compilation Function

In the following table, we present the compilation function. In the first column, we present the example written in Mini [5], in the second, the corresponding AST and finally we present the STALlion program.

Mini Program	AST	STALlion Program
variable=5	ASSIGN variable Intdenot 5	l_1 : Mov $r_{var} \leftarrow 5$
variable=expresion	ASSIGN variable Expresion	l_1 : Mov $r_{var} \leftarrow r_{expr}$
a_array[expresion]=value	SAFE_ASSIG_ARRAY a_array Expresion value	l_1 : StoreArray $r_{arr}[r_{expr}] \leftarrow r_{val}$
a_array[expresion]=value	UNSAFE_ASSIG_ARRAY a_array Expresion value	l_1 : MemoryRead $r_i \leftarrow M[r_{arr}]$ l_2 : Comp $r \leftarrow r_i \quad r_{ind}$ l_3 : bez_if $r \quad l_4 \quad l_6$ l_4 : StoreArray $r_{arr}[r_{ind}] \leftarrow r_{val}$ l_5 : *jump_down* l_6 l_6 : ARRAY ERROR l_7 : ...
var_x=a_array[expresion]	ASSIGN var_x SAFE_ARRAY a_array Expresion	l_1 : LoadArray $r_a \leftarrow r_{arr}[r_{expr}]$ l_2 : Mov $r_x \quad r_a$

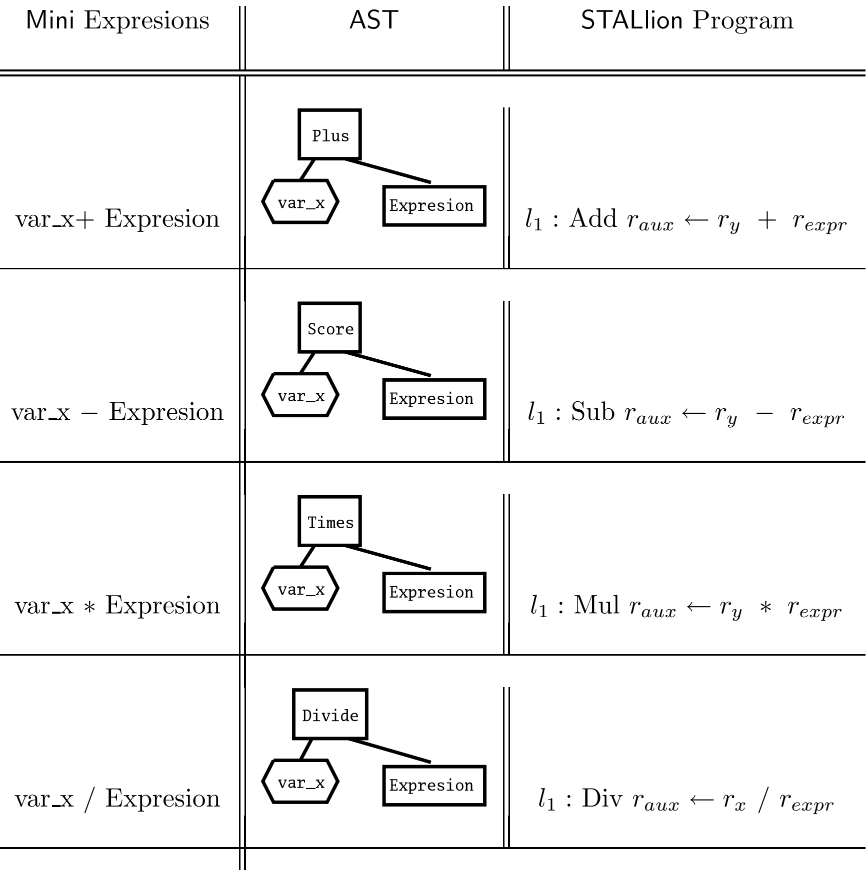
if (var_x>var_y) {
Body_Then
}
else
Body_Else
Sentences
	IF
	Sentences
	Mayor
	var_x
	var_y
	Body_Then
	Body_Else
		l_1 : Comp $r \quad r_x \quad r_y$
		l_2 : bez_if $r \quad l_3 \quad l_5$
		l_3 : Basic_Body_Then
		l_4 : jump_down l_5
		l_5 : Basic_Body_Else
		l_6 : Instructions



Mini Program	AST	STALlion Program

while (var_x>var_y) {
Body_W
}
Sentences
	WHILE, Sentences, Mayor, var_x, var_y, Body_W	l_1 : Comp r r_x r_y
		l_2 bez_while r l_5
		l_3 : Basic_Body_While
		l_4 : jump_up l_1
		l_5 : Instructions

The expresions are compiled by the following way:



Mini Expresions	AST	STALlion Program
var_x+ Expresion	Plus, var_x, Expresion	l_1 : Add $r_{aux} \leftarrow r_y + r_{expr}$
var_x − Expresion	Score, var_x, Expresion	l_1 : Sub $r_{aux} \leftarrow r_y - r_{expr}$
var_x ∗ Expresion	Times, var_x, Expresion	l_1 : Mul $r_{aux} \leftarrow r_y * r_{expr}$
var_x / Expresion	Divide, var_x, Expresion	l_1 : Div $r_{aux} \leftarrow r_x / r_{expr}$

The figure 8 shows the generated STALlion code for the example of the figure 7.

5 Related Work

TAL is an extension of a non-typed assembly language with code annotations, primitives of memory handling and a set of typed rules [14]. The goal of TAL is to provide a static typed object language

l_1 : Mov $r_{index} \leftarrow 0$
l_2: Mov $r_{value} \leftarrow 1$
l_3 : Mov $r_{sum} \leftarrow 0$
l_4: Comp r $\leftarrow r_{index} \quad 10$
  l_5 : bez_while $r \quad l_{10}$
  l_6 : StoreArray $r_{data}[r_{index}] \leftarrow r_{value}$
  l_7 : Add $r_{value} \leftarrow r_{value} \quad 1$
  l_8 : Add $r_{index} \leftarrow r_{index} \quad 1$
  l_9 : Jump_up l_4
l_{10} : Comp $r_2 \leftarrow r_{index} \quad 0$
  l_{11} : bez_while $r_2 \quad l_{17}$
  l_{12} : Add $r_{index2} \leftarrow r_{index} \quad 1$
  l_{13} : LoadArray $r_a \leftarrow r_{data}[r_{index2}]$
  l_{14} : Add $r_{sum} \leftarrow r_{sum} \quad r_a$
  l_{15} : Sub $r_{index} \leftarrow r_{index} \quad 1$
  l_{16} : Jump_up l_{10}
l_{17}: ...

Figure 8: STALlion code for the AST of the figure 7

more suitable than Java bytecodes to support a great variety of source languages and an important number of optimizations. At the moment, TAL is implemented in the Intel IA32 architecture (*32-bit 80x86 flat model*), and is called TALx86.

A TAL program contains assembly code together with type annotations and TAL pseudo-intructions, which are used to check the assembly language program with the type checker.

The typed rules guarantee memory security, control flow security and type security of the TAL programs. TAL is an ideal platform for compilers pointed by types that want to produce safe code for the use of mobile code applications or extending operative systems kernels. The compiler developers can use the TAL program security to "purify" tranformations of sophisticated program. Types help us to check the soundness of the transformations and optimizations that are very difficult to make without types. The TAL applets, similar to java applets, can be downloaded from non-trusted servers from internet, to verify and then execute without damaging of the host security.

DTAL is a dependently typed assembly language [18]. Its type system supports the use of dependent types, obtaining some advantages of dependent type assembly level. DTAL is better than TAL, allowing for important optimizations in the compilers, such as array bound checking. DTAL addresses formally the problem of sum type representation in an assembly level, doing this handling not only data types in ML but also dependent data types in dependent ML (DML).

HBAL is a typed assembly language oriented to ensure that each program of a valid type is executed in a heap bounded memory [4, 7] . If the type of a program HBAL is verifiable, it is executed in a safe form in any memory that satifies the initiated type conditions. In order to guarantee the safe execution, a program P must:

1. check the type of P once,
2. before each execution, check that the memory satisfies the type assumption of initial context

Others TALs (Typed Assembly Languages) base their type systems on a particular configuration of memory, as long as the program type is checked before each execution. The HBAL language ensures, by means of static analysis, that the heap bounded spaces assigned to the program are respected (it is not allowed to change arbitrary places in memory), no reading of not initiated addresses (is not allowing reading of "garbage") and no reading of addresses that the program has not written before(it is not allowed the round of free memory and there are not lose of private information).

HBAL distinguishes explicitly between addresses devoted to store code and devoted to store datas by means of an appropriate discipline of types. It can also eliminate the possibility of re-writing addresses where the code is stored. By adding to this the restriction of control flow by typing the label, is eliminated the danger of execution of hidden instructions on the addresses where dates are stored. As for the verification of non-interference property, it enforces that in the execution of a program, the public information does not depend confidencial information.

6 Conclusions and Future Work

We developed a typed assembly language that allows us to make various kinds of static analysis for the purpose of ckecking security properties. This is a preliminary approach to a mixed check system of security policies, which allows us to verify security properties based on flow analysis and data analysis over a formal type system. The security verifications made on STALlion are pointer alias analysis, array bound checking, initialized values and label checking.

An interesting future work is to define an isomorphism between Mini, AST and STALlion. Furthermore, it could be interesting to continue researching in the typed assembly language area. This should allow us to extend STALlion incorporating new instructions but also defining a memory space that only contains data and another space that only contains instructions. In this way, we should prevent many security violations, for example, the violations done by buffer overflow. We plan to define a classification of which security properties can be done efficiently over STALlion and which over the AST.

Another future work is the incorporation of STALlion to the implemented prototype of safe mobile code environment [13].

References

[1] A. Appel, A. Felty, "A Semantic Model of Types and Machine Instructions for Proof-Carrying Code", in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'00), pp. 243–253, ACM Press, Boston, Massachusetts (USA), January 2000.

[2] A. Appel, "Foundational Proof-Carrying Code", in *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pp. 247–256, IEEE Computer Society Press, 2001.

[3] A. Appel, E. Felten, "Models for Security Policies in Proof-Carrying Code". Princeton University Computer Science Technical Report TR-636-01, March 2001.

[4] D. Aspinall, A. Compagnoni, "Heap Bounded Assembly Language", March 2001, Journal of Automated Reasoning, 2003, vol. 31, iss. 3-4, pp. 261-302(42). Kluwer Academic Publishers. Special issue on Proof-Carrying Code. 2004.

[5] F. Bavera, M. Nordio, J. Aguirre, M. Arroyo, G. Baum, R. Medel, "CCMini: A prototypy of Certifying Compiler". Submitted for publication. 2004.

[6] A. Bernard, P. Lee, "Temporal Logic for Proof-Carrying Code", in *Proceedings of Automated Deduction* (CADE-18), Lectures Notes in Computer Science 2392, pp. 31–46, Springer-Verlag, 2002.

[7] A. Compagnoni, M. Lucotte, R. Medel, "Implementing a Typed Assembly Language and its Machine Model", in Proceedings of the CACIC'02 (Argentinean Conference of Computer Science), Buenos Aires (Argentina), October 2002.

[8] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, K. Cline, "A certifying compiler for Java", in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'00), pp. 95–105, ACM Press, Vancouver (Canada), June 2000.

[9] G. Necula "Compiling with Proofs" Ph.D. Thesis School of Computer Science, Carnegie Mellon University CMU-CS-98-154. 1998.

[10] G. Necula, P. Lee, "The Design and Implementation of a Certifying Compiler", in *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'98), pp. 333–344, ACM Press, Montreal (Canada), June 1998.

[11] G. Necula, R. Schneck, "Proof-Carrying Code with Unstrusted Proof Rules", in *Proceedings of the 2nd International Software Security Symposium*, November 2002.

[12] G. Necula, R. Schneck, "A Sound Framework for Untrusted Verification-Condition Generators", in *Proceedings of IEEE Symposium on Logic in Computer Science* (LICS'03), July 2003.

[13] M. Nordio, F. Bavera, R. Medel, J. Aguirre, G. Baum, " A Framework for Execution of Secure Mobile Code based on Static Analysis". To be published at XXIV International Conference of the Chilean Computer Science Society. IEEE-CS PRESS. 2004.

[14] G. Morrisett, K. Crary, N. Glew and D. Walker, "Stack-Based Typed Assembly Language". In Second International Workshop on Types in Compilation. Kyoto, pp. 95-117. Published in Xavier Leroy and Atsushi Ohori, editors, Lecture notes in Computer Science, volume 1473, pages 28-52. Springer-Verlag, 1998.

[15] M. Plesko, F. Pfenning, "A Formalization of the Proof-Carrying Code Architecture in a Linear Logical Framework", in *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento (Italy), 1999.

[16] Fred B. Scheneider, "Enforceable security policies". Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department, September 1998.

[17] R. Schneck, G. Necula, "A Gradual Aproach to a More Trustworthy, yet Scalable, Proof-Carrying Code", in *Proceedings of International Conference on Automated Deduction* (CADE'02), pp. 47–62, Copenhagen, July 2002.

[18] H. Xi and R. Harper, "A Dependently Typed Assembly Language". Technical Report OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology. 1999.