Information Flow Type System for Proof Carrying Code

Abdulrahman Muthana, Abdul Azim Abd Ghani, Ramlan Mahmod, and Hasan Selamat
ab.muthana@gmail.com {azim, ramlan, hasan}@fsktm.upm.edu.my
Faculty of Computer Science and Information Technology,
Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia

Summary
Proof Carrying Code (PCC) is a promising new technology for enforcing security policies. We present an information flow type system for RISC-style assembly language that enforces confidentiality through noninterference. Based on the security-type system, the PCC can be used for checking untrusted code for noninterference, and thus enabling end-users to protect their confidential data.

Key words:
Information-Flow, Proof Carrying Code, Type system, Static Analysis, Noninterference, Assembly Language.

1. Introduction

Proof Carrying Code (PCC) is a technique developed by Necula and Lee [7] as a safety framework for mobile code and operating system extensions. The trusted computing base (TCB) in PCC is relatively small and easy to implement. Furthermore, to install and execute the untrusted code, the code receiver checks only its safety proof rather than checking the program text, which is more intricate task. These advantages make PCC an active research area see e.g. [2, 3, 4, 5, 6, 7, 8, 9, 10] and an attractive option for enforcing security policies.

A key component of PCC infrastructure is the security policy, which defines the desired security requirements that the untrusted code must meet. In order to adapt PCC to static information-flow analysis, this key component must essentially enforce the confidentiality. The motivation of adapting PCC to static information flow analysis is the need for a reliable mechanism to protect confidential data of end-users from being leaked by the untrusted code.

Recent works in language-based security have shown that type systems can be used as a basis for any security infrastructure wishes to provide an adequate assurance of confidentiality, and in particular through noninterference [25]. The notion of noninterference guarantees the confidentiality through the absence of illegal information flow.

Many works have addressed enforcing confidentiality through noninterference, but unfortunately, most of these studies have been devoted to source-level languages and calculi [11], see e.g. [12, 13, 14, 15, 16, 17, 18, 19]. However, enforcing confidentiality at low-level is highly desirable because (1) ultimately, it is the machine code that executes; (2) much of the code is distributed in low-level form; (3) it yields a small trusted computing base.

This paper presents an information flow type system for RISC-style assembly language as a basis for the security policy of PCC infrastructure to enforce confidentiality through noninterference, and thus enabling end-users to protect their confidential data. The contribution of this paper is that we consider it as a useful step toward enabling PCC to benefit from a large body of work in type-based static information-flow analyses.

The remainder of this paper is organized as follows: section 2 gives a summary of the related work. The assembly language including the syntax, the semantics, and the control dependence regions is described in section 3. Section 4 presents the security policy. In section 5 we give a brief account on PCC for noninterference framework based on the proposed security-type system. Section 6 concludes.

2. Related Work

Many works have addressed enforcing confidentiality through noninterference, but unfortunately, most of these studies have been devoted to source-level languages and calculi [11], see e.g. [12, 13, 14, 15, 16, 17, 18, 19]. In the following, we restrict ourselves to discussing very closely related work; in particular those that studied noninterference at assembly level.

Zdancewic and Myers [20] presented low-level, secure calculus that guarantees noninterference property. To permit high precision information flow analysis they used ordered linear continuations to simulate source-level program structures. Their language is not an assembly language because it has if-then-else structure, has no register file, and is based on variables.

Bonelli et al. [21] presented typed assembly language for secure information flow SIFTAL. Their technique is inspired by the work of Zdancewic and Myers in that they use the notion of linear continuations for implicit flow tracking.

Medel et al. [22] presented SIF language, an improved version of SIFTAL. SIF introduces a stack of junction points for conditionals and uses two pseudo-instructions

Manuscript received June 5, 2007
Manuscript revised July 25, 2007

for handling this stack in order to recover the structured control flow.

Yu and Islam [23] presented TALC language for enforcing noninterference in assembly code. TALC is similar to SIF in that they use type annotations to recover structured control flow of source-level programs and no trusted component for computing postdominators is required. However TALC is richer than SIF. TALC supports code pointers and call stack. [23] Also presented a security type-preserving compilation from source-level security language with first order procedures to TALC.

The mentioned works are based on the philosophy of type-preserving compilation in which the compiler produces the static type annotations, which expressed in the code, to recover the source-level program abstractions. In contrast, our type system is for information flow analysis of assembly code produced by an off-the-shelf compiler that does not produce such code annotations. We circumvent the lack of such annotations by using a trusted function to retrieve the missing source-level abstractions.

Avvenute et al. [24] proposed an approach for verifying secure information flow in java bytecode. The proposed approach is similar to type-level abstract interpretation used in standard Java bytecode verification.

Barthe et al. [1] presented information flow type system for a simplified version of JVM and introduced a security type-preserving compilation from source-level language into their language.

Barthe et al. [26] developed an information flow type system for a fragment of bytecode that extended the JVM language defined in [1] with new features to include classes, objects, and exceptions. They also proved that the information flow type system enforces noninterference.

Barthe et al. [27] presented a formal relation between security policies at source-level language and security policies at bytecode level. This relation is realized via type-preserving compilation. A main point in [27] is to derive an information flow type system for source program from an information flow type system for bytecode.

It should be noted that works [1, 24, 26, 27] address stack-based model (bytecode) which is different from RISC-style architecture.

In his PhD thesis [7], Necula indicated that adapting PCC to information flow analysis of assembly language is an open research area and in [11], Sabelfeld and Myers pointed out that such adapting is highly desirable.

3. Assembly Language

This section introduces a generic assembly language SAL derived from [7] which is used as a basis for describing the proposed type system.

3.1 Syntax

The syntax of SAL is shown in Fig. 1. SAL is RISC-like assembly language, with a finite set of general purpose registers. In addition, SAL has a number of special purpose registers: program counter "pc" that holds the address of current instruction, stack pointer register "sp", and register "*ra*" which used to hold the return address of current function. More details on SAL language can be found in [7].

Register	Regs::=r_i | *ra*	i= 0, …, R_{max}
Instruction:	I ::= $r = n$	Immediate Move
	$r = r'$	Register Move
	r = aop r', n	Arith./Logical Operations
	r = aop r', r''	Arith./Logical Operations
	ra = pc + n	Compute return address
	jump *label*	Jump to label
	jcond *r*, *label*	Conditional branch
	call F	Function call
	ret	Function return
	r = M[r']	Memory read / Load
	M[r'] = r	Memory write / Store
	r = M[sp + n]	Stack read
	M[sp + n] = r	Stack write
	sp = sp + n	Advance stack pointer
Numerals :	$n \in \mathbb{Z}$	

Fig. 1 SAL Instruction set.

3.2 Operational Semantics

SAL execution state *st* is defined as a triple (i, M, R), where i is the value of program counter, M : mem → n is a memory descriptor which is a mapping from memory locations to values, R: Regs → n is a register file which is a mapping from registers to values. i++ refers to the address of next instruction immediately following the current instruction.

SAL program consists of function definitions each of which is a sequence of instructions taken from Fig. 1. Furthermore, each SAL program has a function main. Fig. 2 shows the operational semantics of SAL, giving the resulting state obtained after executing each instruction.

Instruction F(i)	Resulting state
$r = n$	(i++, M, R[$r \leftarrow n$])
$r = r'$	(i++, M, R[$r \leftarrow$ R(r')])
r = aop r', n	(i++, M, R[$r \leftarrow$ (R(r') aop n)])
r = aop r', r''	(i++, M, R[$r \leftarrow$ (R(r') aop R(r''))])
r = M[r']	(i++, M, R[$r \leftarrow$ M(R(r'))])
M[r'] = r	(i++, M[(R(r')) $\leftarrow$ R(r)], R)
r = M[sp+ n]	(i++, M, R[$r \leftarrow$M(R(sp)+n)])
M[sp + n] = r	(i++, M[(R(sp) + n) $\leftarrow$R(r)], R)
sp = sp + n	(i++, M, R[sp $\leftarrow$(R(sp) + n)])
jump *label*	(*label*, M, R)
jcond r , *label*	(*label*, M, R)
jcond r , *label*	(i++, M, R)
call G	(<G,0>, M, R) , ra = i++
ret	(R(ra), M, R)

Fig. 2 Operational Semantics of SAL.

3.3 Control Dependence Regions

Conditional branch instructions are the source of implicit flow. They control the execution of other instructions based on value of conditional expressions. When a conditional expression is evaluated, a branch is chosen accordingly and its instructions are executed. Hence, the value of the expression could be inferred by observing the effects of the executed instructions.

To detect implicit flow, we must first identify for each conditional instruction the set of instructions that execute under its control condition. The set of these instructions constitutes what is called the control dependence region, *CDR*. Every conditional instruction has a control dependence region.

We use the notion of control flow graph and the notion of postdomination to identify control dependence regions. The body of a given function F consists of set of basic blocks B, denoted as BB_F. The control flow graph of function F is a directed graph (V, E), where V= BB_F the set of nodes, and E $\subseteq$ V x V, the set of edges. The control flow graph is augmented with two additional nodes *start* and *exit*. There is an edge from *start* node to the first node of control flow graph and an edge from *start* node to *exit* node. Furthermore, there is an edge from every *return* node to *exit* node.

Let B_i, $B_j \in$ V:

B_j postdominates B_i, denoted by B_j=pdom(B_i), if $B_i \neq B_j$ and B_j is on every path from B_i to *exit* node.

B_j immediately postdominates B_i, denoted by B_j=ipdom(B_i), if $B_i \neq B_j$ and there is no node B_k such that B_k=pdom(B_i) and B_j=pdom(B_k).

An edge (B_i, B_j) of set E means that, the instructions of B_j are immediately executed after that of B_i. CDR_i is the set of basic blocks whose instructions are executed conditionally based on the value of expression tested at address i. Thus, CDR_i constitutes the control dependence region associated with the conditional branch instruction at address i.

Type system is parameterized with abstract functions: *region*, *ipd*, and *propagate*. The function *region(i)* identifies the control dependence region of a conditional branch instruction at address i. The function *ipd*(i) returns the address of the instruction that is immediately executed after exiting from *region*(i), thus, representing the immediate postdominator of instruction at address i. Finally, the function *propagate (region, security level)* updates the security context of instructions of a given region into a given security level. In addition, we use the following functions: *addr, ctxt, first, and dom.* The function addr(i) returns true if i is an instruction address, the function *ctxt*(i) denotes the security context of an instruction i, the function *first*(B) returns the first instruction address in basic block B, and finally the function *dom*(U) returns the instruction addresses that belong to a given region U :

In the following we give a formal definition for the functions *region*, *ipd*, and *propagate*:

region(i) = $\forall$j. *addr*(j) and j $\in$ *dom*(CDR_i).

ipd(i) = $\exists$j. *addr*(j) and j= *first*(B_m) and $i \in$ *dom*(B_n) and B_m=ipdom (B_n).

propagate (region(i), ℓ) = $\forall$j. *addr*(j) and j $\in$ *region*(i) implies *ctxt*(j) = ℓ.

If i and j are two conditional instructions addresses such that j $\in$ CDR_i, then $CDR_j \subset CDR_i$ meaning that control dependence region associated with j is included in that associated with i; this is called region inclusion property, RIP.

4. Security Policy

The main goal of our security policy is to guarantee secure information flow within SAL programs. The main components of the security policy are information flow type system and logic. The information flow type system, shown in Fig. 3, is a set of typing rules for information flow analysis of SAL programs that enforces confidentiality through noninterference.

$$[\mathrm{T}_{\mathrm{mvi}}]\ \frac{\mathrm{F(i)}\coloneqq\ r=n \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma\{r:\ell_{\mathrm{pc}}\}}{\Gamma;\ \mu\vdash r=n}$$

$$[\mathrm{T\_mov}]\ \frac{\mathrm{F(i)}\coloneqq\ r\ =\ r' \qquad \Gamma(r'):\ell' \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \qquad \Gamma\{r:\ell_{\mathrm{pc}}\ \sqcup\ \ell'\}}{\Gamma;\ \mu\ \vdash\ r\ =\ r'}$$

$$[\mathrm{T\_aop\_n}]\ \frac{\mathrm{F(i)}\coloneqq\ r\ =\ \mathrm{aop}\,r',n \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma(r'):\ell' \quad \Gamma\{r:\ell_{\mathrm{pc}}\ \sqcup\ \ell'\}}{\Gamma;\ \mu\ \vdash\ r\ =\ \mathrm{aop}\,r',n}$$

$$[\mathrm{T\_aop\_r}]\ \frac{\mathrm{F(i)}\coloneqq\ r\ =\ \mathrm{aop}\,r',r'' \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma(r'):\ell' \quad \Gamma(r''):\ell'' \quad \Gamma\{r:\ell_{\mathrm{pc}}\sqcup\ell'\sqcup\ell''\}}{\Gamma;\ \mu\ \vdash\ r\ =\ \mathrm{aop}\,r',r''}$$

$$[\mathrm{T\_cond}]\ \frac{\begin{array}{c}\mathrm{F(i)}\coloneqq\ \mathrm{jcond}\ r,\ label \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma(r):\ell \\ propagate\big(region(i),\ell_{\mathrm{pc}}\sqcup\ell\big) \\ \mathrm{j}=\mathrm{ipd(i)} \quad \mathrm{j}:<\Gamma'> \qquad \Gamma(\mathrm{pc})=\Gamma'(\mathrm{pc}),\ \ \Gamma'\subseteq\Gamma\end{array}}{\Gamma;\ \mu\ \vdash\ \mathrm{jcond}\ r,label}$$

$$[\mathrm{T\_load}]\ \frac{\mathrm{F(i)}\coloneqq r=\mathrm{M}[r'] \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma(r'):\ell' \quad \mu([r']):\ell_{\mathrm{m}} \qquad \Gamma\{r:\ell_{\mathrm{pc}}\sqcup\ell'\sqcup\ell_{\mathrm{m}}\}}{\Gamma;\ \mu\ \vdash\ r\ =\ \mathrm{M}[r']}$$

$$[\mathrm{T\_store}]\ \frac{\mathrm{F(i)}\coloneqq \mathrm{M}[r']=r \quad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma(r):\ell \quad \Gamma(r'):\ell' \quad \mu([r']):\ell_{\mathrm{m}} \quad \ell_{\mathrm{pc}}\sqcup\ell\sqcup\ell'\sqsubseteq\ell_{\mathrm{m}}}{\Gamma;\ \mu\ \vdash\ \mathrm{M}[r']=\ r}$$

$$[\mathrm{T\_call}]\ \frac{\begin{array}{c}\mathrm{F(i)}\coloneqq\ \mathrm{call\ G} \quad \Sigma_{\mathrm{G}}\ =\ (Pre,Post,Sig) \\ Pre_{\mathrm{G}}\ =\ r_1:\ell_1\ \wedge\ r_2:\ell_2\ \wedge\ \ldots\ \wedge\ r_{\mathrm{n}}:\ell_{\mathrm{n}} \\ \Gamma(Pre_{\mathrm{G}}):\{r_1:\ell_{\mathrm{a1}}\ \wedge\ r_2:\ell_{\mathrm{a2}}\ \wedge\ \ldots\ \wedge\ r_{\mathrm{n}}:\ell_{\mathrm{an}}\} \\ \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \ell_{\mathrm{ai}}\sqcup\ell_{\mathrm{pc}}\sqsubseteq\ell_{\mathrm{i}} \quad \forall\,\mathrm{i}\in\{1\ldots\mathrm{n}\}, \quad \ell_{\mathrm{pc}}\sqsubseteq\mathrm{Sig}\end{array}}{\Gamma;\ \mu\ \vdash\ \mathrm{call\ G}}$$

$$[\mathrm{T\_jump}]\ \frac{\mathrm{F(i)}\coloneqq\ \mathrm{jump}\ \ label}{\Gamma;\mu\vdash\ \mathrm{jump}\ \ label}$$

$$[\mathrm{T\_ret}]\ \frac{\begin{array}{c}\mathrm{F(i)}\coloneqq\ \mathrm{ret} \qquad \Sigma_{\mathrm{F}}\ =\ (Pre,Post,Sig) \\ Post_{\mathrm{F}}\ =\mathrm{r}:\ell_{\mathrm{post}} \quad \Gamma(Post_{\mathrm{F}})\ =\ \{\mathrm{r}:\ell_{\mathrm{ret}}\} \quad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \\ \ell_{\mathrm{ret}}\ \sqcup\ell_{\mathrm{pc}}\sqsubseteq\ \ell_{\mathrm{post}}\end{array}}{\Gamma;\ \mu\ \vdash\ \mathrm{ret}}$$

$$[\mathrm{T\_sstore}]\ \frac{\mathrm{F(i)}\coloneqq \mathrm{M}[\mathrm{sp}+n]=r \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma(r):\ell \quad \Gamma\{(\mathrm{sp}+n):\ell_{\mathrm{pc}}\sqcup\ell\}}{\Gamma;\ \mu\ \vdash\ \mathrm{M}[\mathrm{sp}+n]=r}$$

$$[\mathrm{T\_sload}]\ \frac{\mathrm{F(i)}\coloneqq r=\mathrm{M}[\mathrm{sp}+n] \qquad \Gamma(\mathrm{pc}):\ell_{\mathrm{pc}} \quad \Gamma(\mathrm{sp}+n):\ell_{sp+n} \quad \Gamma\{r:\ell_{\mathrm{pc}}\sqcup\ell_{sp+n}\}}{\Gamma;\ \mu\ \vdash\ r=\ \mathrm{M}[\mathrm{sp}+n]}$$

Fig. 3 Typing rules of information flow analysis of function F.

To formalize the type system, we assume a lattice $\mathcal{L}$ of security levels, partially ordered by $\sqsubseteq$, with top element $\top$, bottom element $\bot$, and join operation $\sqcup$. To simplify the presentation, we assume that security lattice $\mathcal{L}$ contains a set of two security levels, $\mathcal{L}$= {*L*, *H*} and that $L \sqsubseteq H$; *L* stands for low security data and *H* for high security data.

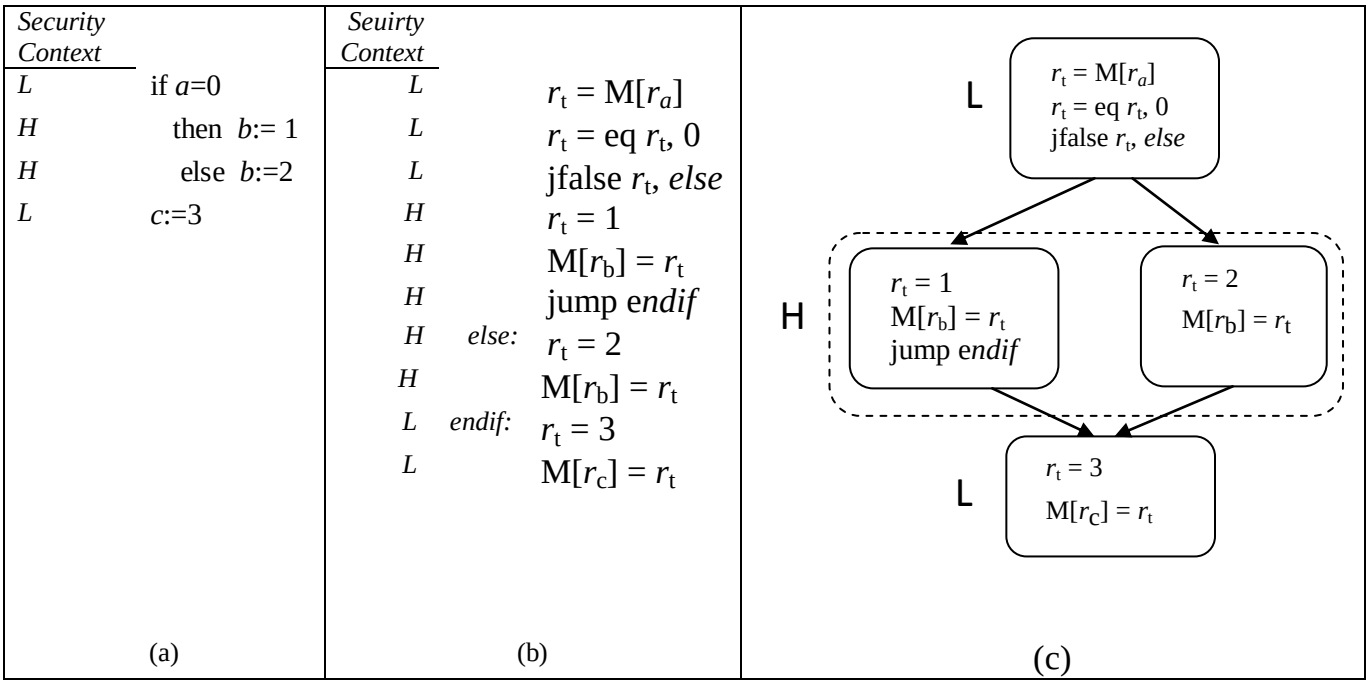

Fig. 4 An example fragment of source program (a) and the corresponding SAL code (b) and the control flow graph showing control dependence region of the conditional enclosed by dashed lines box.

The type system is formalized at assembly level; therefore, it is defined in terms of registers, memory locations, and immediate values. The security levels of registers and memory locations are described in register file type $\Gamma: r \rightarrow \mathcal{L}$ and memory type μ: mem $\rightarrow \mathcal{L}$ respectively. Stack slots are treated as part of register file, and hence described by Γ; we use Γ(pc) to denote the current security context. As an immediate value is not intrinsically sensitive [15]; an immediate value is given a security level *L*.

To prevent explicit flow, the type system prevents high security values from flowing to lower memory locations and to prevent implicit flow, the type system associates a security level with program counter pc at each program point, which is called the security context, and checks that the security level of updated data is as at least as the security context.

Preventing illegal information flow through function calls and returns requires that each function to be having a typing specification. The typing specification of a given function F, Σ_F, is a triple (*Pre, Post, Sig*), where *Pre* is the precondition, *Post* is the postcondition, and *Sig* is the signature of function F. For any given function, the precondition represents binding of its input parameters with security levels, the postcondition represents binding of its return value with a security level, and the signature represents binding of its name with a security level. The signature *Sig* of a given function F represents the lowest security level of data that function F may update.

Typing rules T_mov, T_mvi, T_aop_n, T_aop_r, T_load, T_sload, and T_sstore infer the security levels of destination registers from the source operands taking into account the current security context.

In the rule T_store, the condition is that the security level of the target memory location is higher than or equal to those of the source operand, the address register, and the current security context.

The rule T_cond performs least upper bound between the current security context and the security level of register *r* that controls the branching. The result security level then propagated through the region of the conditional instruction as a security context for all instructions in the region in order to prevent implicit flow. Moreover, the rule T_cond checks that the security context at the postdominator j matches the current security context. To ensure a precise information flow analysis in the remainder of the code, the rule T_cond requires that the register file types at j, Γ', complies with the current register file types, Γ; this is checked through subtyping relation, $\subseteq$.

The rule T_call checks that, for any given function, the security levels of the formal parameters as specified by the user in the precondition *Pre* are higher than or equal to those of the corresponding actual parameters and the current security context. In addition, it checks that the signature *Sig* is higher than the current security context.

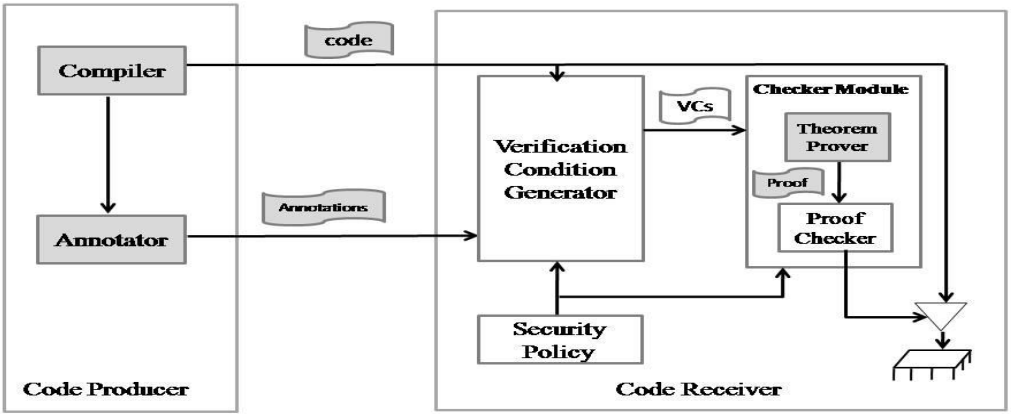

Fig. 5 The high-level structure of the PCC for noninterference Framework.

Similarly, the rule T_ret checks that the security level specified by the user for the function return value in postcondition *Post* is higher than or equal to those of the actual return value and the current security context.

Example*:** consider a fragment of program, shown in Fig.4.a, and its corresponding SAL code, shown in Fig. 4.b, where ***a is a high variable, ***b*** and ***c*** are low variables. Memory locations $\mathbf{M}[r_a]$, $\mathbf{M}[r_b]$, $\mathbf{M}[r_c]$ in SAL code correspond to variables ***a***, ***b***, and ***c*** respectively. Hence, $\mathbf{M}[r_a]$ is high memory location; $\mathbf{M}[r_b]$ and $\mathbf{M}[r_c]$ are two low memory locations. The program modifies the value of variable ***b*** based on the value of variable ***a***. In this fragment, an implicit flow occurs because the value of high variable ***a*** can be inferred from the final value of low variable ***b***. Our type system can prevent such leakage by propagating the high security level through the control dependence region of the conditional, as shown in Fig 4.c, and by rejecting writing into low memory location $\mathbf{M}[r_b]$ when the current security context is high.

5. Proof Carrying Code for Noninterference

Fig. 5 shows a source-level structure of PCC for noninterference framework. Modules are the *Compiler*, the *Annotator*, the *Security policy*, the *Verification Condition Generator (VCG)*, and the *Checker Module*, which includes the *Theorem Prover* and the *Proof Checker*. Following the conventions used in [7], grey rectangular boxes represent the untrusted modules, and white ones represent the trusted modules that constitute *Trusted Computing Base (TCB)*. In the following we give a brief description of each module.

The compiler is an off-the-shelf compiler.

The Annotator produces the typing specifications and initial annotations that ***VCG*** requires to verify the code. The initial annotations are bindings of security levels with global objects in addition to functions' typing specifications.

The Security Policy consists of logic and an information flow type system for assembly language. The logic is a set of first order predicate constructors, axioms, and proof rules designed to formalize functions typing specifications, construct verification conditions, and guide the theorem proving process. The type system, shown in Fig. 3, is a set of typing rules that are used for information flow analysis of SAL programs. The type system is parameterized by control dependence regions computed by a trusted function, which performs intra-procedural control flow analysis of the untrusted code.

The Verification Condition Generator (VCG) performs an abstract execution over the untrusted code based on the type system and typing specifications one function at a time. *VCG* produces verification conditions (VCs) for memory write, function call, and function return instructions. VCs and their assumptions are represented as LF terms [28].

The Checker Module includes the theorem prover and the proof checker. The Twelf system [29] is our checker module. The object logic is encoded in Twelf and then the Twelf theorem prover generates the proofs that are to be type checked later on by Twelf Type checker.

6. Conclusion

We have proposed an information flow type system for RISC-style assembly language that can serve as a basis for the security policy of PCC infrastructure to enforce confidentiality through noninterference. The use of PCC for checking untrusted code for noninterference based on the proposed type system will enable end-users to protect their confidential data. We consider this as a useful step toward enabling PCC to benefit from a large body of work in type-based static information-flow analyses.

References

[1] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *LNCS*, pages 2–15, Venice, Italy, Jan. 2004.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] R. Schneck, G. Necula, "A Gradual Approach 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.

[11] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21:5-19, January 2003.

[12] F. Pottier and V. Simonet, "Information flow inference for ML," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2002, pp. 319–330.

[13] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *J. Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[14] N. Heintze and J. G. Riecke, "The SLam calculus: programming with secrecy and integrity," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1998, pp. 365–377.

[15] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, LNCS, pages 607–621, Lille, France, Apr. 1997.

[16] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1999, pp. 228–241.

[17] G. Barthe and B. Serpette, "Partial evaluation and non-interference for object calculi," in *Proc. FLOPS*. Nov. 1999, vol. 1722 of *LNCS*, pp. 53–67, Springer-Verlag.

[18] S. Zdancewic and A. C. Myers, "Secure information flow and CPS," in *Proc. European Symposium on Programming*. Apr. 2001, vol. 2028 of *LNCS*, pp. 46–61, Springer-Verlag.

[19] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a Java-like language," in *Proc. IEEE Computer Security Foundations Workshop*, June 2002, pp. 253–267.

[20] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2– 3):209–234, Sept. 2002.

[21] E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A typed assembly language for secure information flow analysis. Technical report, Stevens Institute of Technology, Hoboken, NJ, July 2004.

[22] R. Medel, A. Compagnoni, and E. Bonelli. Non-interference for a typed assembly language. In *Proc. 2005 Workshop on Foundations of Computer Security*, Chicago, IL, June 2005.

[23] D. Yu, N. Islam. A Typed Assembly Language for Confidentiality. Technical report, DoCoMo USA Labs**,** 2005**.**

[24] M. Avvenuti, C. Bernardeschi, and F. Francesco. Java bytecode verification for secure information flow. ACM SIGPLAN Notices,38(12):20–27, Dec. 2003.

[25] J. A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the Symposium on Security and Privacy*, pages11–20. IEEE Press, 1982.

[26] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In TLDI '05: *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[27] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Proceedings of Symposium of Security and Privacy'06*. **IEEE Press**, 2006.

[28] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery* 40(1):143–184, January 1993.

[29] F. Pfenning and C. Sch¨urmann. System description: Twelf: A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16-99)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10 1999. Springer.

Abdulrahman Muthana received the B.Sc. degree in Computer Science from Mosul Univeristy, Iraq in 1996 and M.C.A. degree from Bangalore University, India in 2003. In 1997 he joined Thamar University as a lecturer. He is a Ph.D. student at University Putra Malaysia. His research interest is Language-based security.

Abdul Azim Abd Ghani received the B.Sc in Mathematics/Computer Science from Indiana State University in 1984 and M.Sc in Computer Science from University of Miami in 1985. He joined Universiti Putra Malaysia in 1985 as a lecturer in Computer Science. He received the Ph.D in Software Engineering from University of Strathclyde in 1993. He is an Associate Professor and the Dean of Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. His research interests are software engineering, software measurement, software quality, and security in computing.

Ramlan Mahmod hold a PhD from University of Bradford, United Kingdom. Currently, he is a Associate Professor at Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, His research area are artificial intelligence.

Mohd Hasan Selamat received his M.S degrees from Essex University and PhD from East Anglia University in United Kingdom. His research interests including software engineering and information system. He is now a fulltime lecturer and Head Department of Information System in Faculty of Computer Science and Information Technology, University Putra Malaysia.