Journal of Computer Science 5 (2): 163-171, 2009
ISSN 1549-3636

Proof-Carrying Code Based Tool for Secure Information Flow of Assembly Programs

[1]Abdulrahman Muthana, [2]Abdul Azim Abd Ghani, [1]Ramlan Mahmod and [2]Hasan Selamat
[1]Malaysian Institute of Microelectronic Systems Berhad,
Technology Park Malaysia, Kuala Lumpur 57000, Malaysia
[2]Faculty of Computer Science and Information Technology,
University Putra Malaysia, Serdang, Selangor 43400, Malaysia

Abstract: Problem statement: How a host (the code consumer) can determine with certainty that a downloaded program received from untrusted source (the code producer) will maintain the confidentiality of the data it manipulates and it is safe to install and execute. **Approach:** The approach adopted for verifying that a downloaded program will not leak confidential data to unauthorized parties was based on the concept of Proof-Carrying Code (PCC). A mobile program (in its assembly form) was analyzed for information flow security based on the concept of proof-carrying code. The security policy was centered on a type system for analyzing information flows within assembly programs based on the notion of noninterference. **Results:** A verification tool for verifying assembly programs for information flow security was built. The tool certifies SPARC assembly programs for secure information flow by statically analyzing the program based on the idea of Proof-Carrying Code (PCC). The tool operated directly on the machine-code requiring only the inputs and outputs of the code annotated with security levels. The tool provided a windows user interface enabling the users to control the verification process. The proofs that untrusted program did not leak sensitive information were generated and checked on the host machine and if they are valid, then the untrusted program can be installed and executed safely. **Conclusion:** By basing proof-carrying code infrastructure on information flow analysis type-system, a sufficient assurance of protecting confidential data manipulated by the mobile program can be obtained. This assurance was come due to the fact that type systems provide a sufficient guarantee of protecting confidentiality.

Key words: Proof-carrying code, secure information flow, assembly language, non-interference

INTRODUCTION

Recent years have witnessed a growing interest of information flow security analysis due to their connection to the problem of protecting confidential data. The confidentiality policy concerns multi-level security systems. It states that secret data must be protected during the computation and there should be no leakage of that data through public output channel.

Information flow security is formalized as non-interference, which states that final values of low-security variables must be independent of initial high-security variables[1]. Information flow security analysis verifies if a program respects certain confidentiality policy. Denning and Denning[2] were first to perform static information flow analysis for checking programs for confidentiality.

Unfortunately, despite a long history, relatively less interest has been given to low-level languages[3]. Some prevalent trends in software call for techniques to certify machine-code for secure information flow. Among them, dynamic extensibility, where a trusted computing system is extended by importing and executing untrusted mobile code. For example, web browsers plug-ins and operating systems extensions.

Some of research works have studied secure information flow in low-level languages. A low-level, secure calculus that use linear continuations guarantees non-interference property is presented in[6]; however the language is not an assembly language as it has if-then-else structure and has no registers. Recent research works[8-10] studied extending Typed Assembly Language (TAL) with information flow property in order to enforce non-interference in RISC-style assembly programs.

Information flow security analysis of Java bytecode has been studied by several authors. An information flow type system for a simplified version of

Corresponding Author: Abdulrahman Muthana, Malaysian Institute of Microelectronic Systems Berhad, Technology Park Malaysia, Kuala Lumpur 57000, Malaysia

JVM language is developed in[11] and later for an extended fragment of JVM language in[12]. Besides[13], proposed a technique for information flow analysis of Java bytecode using Boolean functions. Methods based on model checking are used for certifying Java applets[14] and a refined technique of[14] is presented in[15] for a subset of JVML. In[16] an approach is proposed for information flow analysis of Java bytecode similar to type-level abstract interpretation used in standard Java bytecode verification and a tool for Java bytecode verification for secure information flow is developed. In[17], the authors developed a semantics based tool for information flow analysis of tack-based assembly language.

This study shows how to determine statically whether it is safe to install and execute a downloaded untrusted code on a host machine. It presents a security analysis technique for certifying assembly programs generated by off-the-shelf compilers for secure information flow based on the concept of Proof-Carrying Code (PCC)[4]. The downloaded code is analyzed based on an information flow type system and security conditions are generated. Proofs of the security conditions are then generated and checked on the host machine by the code consumer's system. If the proofs are valid, the untrusted code can be installed and executed safely.

The researches[8-10] assume existence of special compilers (certifying compilers) that generate the target assembly languages through a trusted compilation. In contrast, the proposed security technique operates directly on assembly language programs generated by general-purpose off-the-shelf compilers. Moreover, abovementioned techniques do not produce explicit proofs for the programs acceptable by such techniques. The most prominent difference between the proposed security analysis technique and techniques of[11-17] is that they treat a stack-based assembly language which is much different from RISC architecture. Furthermore, none of the works[11-17] produces explicit machine-checkable proofs of confidentiality conformance.

Proof-carrying code for secure information flow: The security analysis technique we adopt to verifying secure information flow is based on the concept of proof-carrying code[4]. The information flow security analysis is divided into four phases: developing information flow policy, generating the verification conditions, generating the proofs and checking the proofs. The proposed security analysis techniques consider RISC-style assembly language, SAL[4].

First, information flow analysis type system that defines the authorized information flows within SAL assembly programs and serves as the basis of the security analysis technique is defined[5]. Then we 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[19]. The body of a given function F consists of a set of basic blocks B, denoted as BB_F. The control flow graph of function F is a directed graph (N, E), where $N = BB_F$ the set of nodes and $E \subseteq N \times N$, the set of edges. For two basic blocks B_i and B_j we say that 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 and that B_j immediately postdominates B_i, denoted by B_j = ipd(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).

The security 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. We use a stack, which is called immediate postdominator IPD, to handle implicit information flow by storing the regions and their immediate postdominators.

Second, the verification condition generator VCG executes the assembly program P abstractly (operating on security levels instead of actual values) one function at a time based on typing rules of the security type system. Each class of instructions has a corresponding rule and the VCG builds an abstract state obtained by abstractly executing of an instruction by applying of its rule. The state of abstract executor is defined by a triple AE(i, σ, so), where i is the value of program counter referring to the instruction to be executed next, σ: R→L is an abstract register state which is a mapping from registers names to security levels from a security lattice L; σ(pc) is the security context of current instruction and so is the stack offset.

The abstract executor starts executing the body of function F with an initial state:

$$AE_F(i_0, \sigma_0, so_0)$$

Where:

F = The function being executed

i_0 = 0, the value of program counter referring to the first instruction in function F

σ_0 = Pre_F, the initial abstract register state, initialized to the security levels as specified by precondition, Pre_F

so_0 = Arg_F, the stack offset is initialized with the arguments of function F, Arg_F

$\sigma_0(pc)$ = Sig_F, the security context is initialized to the security level assigned to function F, Sig_F. The security level, Sig_F, is initially propagated as a security context for all instructions in function F

IPD stack is empty.

For arithmetic and logical operations, memory reading instruction (load) and instructions of moving data between registers or between registers and stack locations the abstract executor updates abstract register state by mapping the destination register into the least upper bound of the security levels of source operands taking into accounts the current security context. In the case of conditional branch, both branches are abstractly executed. The least upper bound of the security levels of the conditional register and the current security context is propagated through the conditional region as a security context for all instructions that are executed under the control of conditional expression. For memory write, function calls and functions returns instructions, the VCG constructs a proper verification condition and sends it to the theorem prover to verify. The verification conditions are encoded as LF terms[20].

Third, to prove the verification conditions generated by the VCG we use a theorem prover for first-order predicate logic, which is also able to generate their detailed proofs. Based on the logic that the host machine specifies, the theorem prover attempts to prove the verification conditions emitted by the VCG and generates their detailed proofs. Theorem proving process is guided by a logic program that describes the absence of illegal information flows in the assembly code. The logic program includes a set of proof rules, which are logical translation of security typing rules.

Fourth, after proving the verification conditions by the theorem prover, their proofs are validated by a proof checker for validation. The proof checker component verifies that the proofs generated by the theorem prover are indeed valid and pertaining to the verification conditions generated by the VCG. If the proofs are valid then the untrusted program is considered secure and can be installed and executed safely on the host machine.

Information flow model: We assume a two-point security lattice L = {Low, High}, partially ordered by $\sqsubseteq$, where Low $\sqsubseteq$ High; Low stands for low-security data and High for high-security data. A program P is defined as a pair (I, V) where I denotes instructions and V denotes the variables. V is partitioned into a set of low-security variables V_{Low} and a set of high-security variables V_{High}.

Non-interference: A program P = (I, V) is secure if, starting form two initial memories which agree on the values of the variables V_{Low}, the program P terminates with two final memories which agree on the values of the variables V_{Low}.

The above definition ensures that the final values of low-security variables are independent of the initial values of high-security variables. Secure information flow is formalized as non-interference[1], which states that the values of high-security do not interfere (affect) the values of low-security variables.

MATERIALS AND METHODS

A tool was developed to demonstrate the practicality of the proposed security system. The tool certifies SPARC assembly programs for secure information flow. The following software and tools are used. Visual Basic 6: A programming tool, which is used to develop the prototype implementation of the proposed security system. GNU GCC C Compiler[7]: compiles C programs into executables for SPARC platforms. Disassembler: Disassembles the executable file produced by the GCC compiler into SPARC assembly language. An alternative option to produce SPARC assembly program directly is to use the option-S with GCC compiler. Emacs Text Editor: edits the Twelf signature, which encodes the object logic. Twelf System: A tool for experimentation in the theory of programming languages and logics[18]. It relies on LF type theory and the principle of judgments-as-types for specifications.

RESULTS

The main result of this study is a security technique for verifying assembly programs for secure information flow. To make all the components and concepts of the proposed security technique more concrete, a tool, which is called SPARC PCC-SIF, was developed for verifying SPARC assembly programs for secure information flow based on the proposed security approach. The tool enables the code consumer to ensure that only the programs that satisfy the confidentiality policy are allowed to execute. A satisfaction of confidentiality means that a program has secure information flow. The information flow property is

formalized based on the notion of non-interference. Thus, the purpose of the tool is to check whether the received program has non-interference property.

Most important high-level characteristics of the proposed tool are (1) It operates directly on machine-code; (2) It enforces information flow policy, namely non-interference policy on SPARC assembly programs; (3) It generates explicit machine-checkable security proofs (certificate) for SPARC assembly programs that are proved secure.

The tool is thus can be regarded as an instance of the security analysis for checking secure information flow of SPARC programs. All components of the security analysis technique are adapted to work on SPARC assembly language. The adapting is a straightforward and quite easy since both SAL and SPARC are RISC architectures.

Table 1 shows the translation from SAL to SPARC. Obviously, there is a one to one correspondence between instructions of SAL and SPARC, which facilitates adapting the security technique for SPARC. The instance of SAL has 32 general purpose registers mapped to SPARC registers. In addition the register r_{15} (%O_7) in SPARC is mapped to SAL register "ra". In 32-bit instance of SAL the base values range between -2^{15} to $2^{15}-1$ and in 64-bit instance of SAL the base values range between -2^{63} to $2^{63}-1$.

A program P is a sequence of SPARC instructions I, P = <I>, consisting of functions each of which is a sequence of SPARC instructions. Furthermore, each program P has a function "main". The operational semantics of SPARC language is defined as a triple (I, M, R), where I is the value of the program counter, M is the memory representing the state of memory locations (program variables) and R represents the current state of registers including stack locations (local variables).

Table 1: The translation table from SAL to SPARC architecture

SAL	SPARC
r = n	mov n, r
r = r'	mov r', r
$r_2 = r_1$ add n	add r_1, n, r_2
$r_3 = r_1$ add r_2	add r_1, r_2, r_3
jump label	jmp label
r = r_1 eq n	cmp r_1, n
jfalse r, label	bne label
r = r_1 eq r_2	cmp r_1, r_2
jfalse r, label	bne label
ra = pc+1 call F	call F
ret	ret
r = M[r']	ld [r'], r
M[r'] = r	st r, [r']
r = M[sp+n]	ld [sp+n], r
M[sp+n] = r	st r, [sp+n]
sp = sp+n	save sp, n, sp

The operational semantics is given in terms of resulting state obtained after executing each instruction. It is clear that SAL and SPARC language are also semantically equivalents.

Figure 1 shows the high level structure of the tool. It consists of a number of components: Control Dependence Region Calculator, Verification Condition Generator (VCG), Checker module, which includes Theorem Prover and Proof Checker.

Control dependence regions calculator: Extracts functions, identifies basic blocks, performs intraprocedural control flow analysis, computes control dependence regions CDR for conditional branches and stores the information about CDR in IFD table.

Verification condition generator module: Performs an abstract execution on the code based on the typing rules and initial annotations one function at a time. The verification condition generator begins the execution of the program code starting from function "main". The execution continues until the return instruction is encountered or the VCG reaches an already executed instruction. For each instruction, the VCG builds an abstract state. VCG produces verification conditions for the actions: function calls and returns, memory write instructions. The verification conditions and their assumptions are represented as LF terms and saved in a file.

Checker module: The Twelf system[18] is the checker module. The object logic is encoded as an LF signature. The signature is loaded along with a verification condition file produced by the verification condition generator module. The Twelf theorem prover generates the proof's derivations that are to be type-checked later on by Twelf type-checker. If all proofs are well typed, untrusted program can be executed safely in the code consumer's computing system.

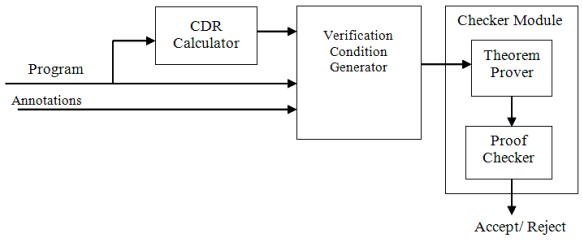

Fig. 1: High-level structure of secure information flow tool

```
int main()
{

if(i==0)
{
 if(k==10)
   m=0;
    else
   m=1;
 }
else
{
  if(j==0)
    k=1;
   else
    k=2;
 }

i=0;
return m;
}
```

(a)

```
main:   save    %sp, -112, %sp
        sethi   %hi(i), %o1
        or      %o1, %lo(i), %o0
        ld      [%o0], %o1
        cmp     %o1, 0
        bne     .LL2
        sethi   %hi(k), %o1
        or      %o1, %lo(k), %o0
        ld      [%o0], %o1
        cmp     %o1, 10
        bne     .LL1
        sethi   %hi(m), %o1
        or      %o1, %lo(m), %o0
        st      %g0, [%o0]
        b       .LL4
.LL1:   sethi   %hi(m), %o1
        or      %o1, %lo(m), %o0
        mov     1, %o1
        st      %o1, [%o0]
.LL2:   sethi   %hi(j), %o1
    or  %o1, %lo(j), %o0
        ld      [%o0], %o1
        cmp     %o1, 0
        bne     .LL3
        sethi   %hi(k), %o1
        or      %o1, %lo(k), %o0
        mov     1, %o1
        st      %o1, [%o0]
        b       .LL4
.LL3:   sethi   %hi(k), %o1
        or      %o1, %lo(k), %o0
        mov     2, %o1
        st      %o1, [%o0]
.LL4:   sethi   %hi(i), %o1
        or      %o1, %lo(i), %o0
        st      %g0, [%o0]
        sethi   %hi(m), %o1
        or      %o1, %lo(m), %o0
        ld      [%o0], %o1
        mov     %o1, %i0
        ret
        restore
```

(b)

```
FuncName: main
signature: Low
precondition : -
postcondition:  %o0: Low
global_section: i : Low, j : Low,k : High, l : Low
                m : Low
```

(c)

Fig. 2: (a): An example source program; (b): The corresponding SPARC program; (c): The typing specification

SPARC PCC-SIF makes several assumptions about the target programs. For example, SPARC PCC-SIF assumes that: (1) Program behavior is insensitive to null operations e.g., no-ops instructions; (2) The last instruction in the program is return instruction; (3) Calling conventions. For example a caller function passes parameters to the called function through a set of registers %o0-%o6 and receives the result value through %o0 register. A register %o7 is reserved for the return address.

The inputs of SPARC PCC-SIF are two text files: the SPARC assembly program file and the specifications file. The source C program (Fig. 2a) consists of one function, main. The program modifies the value of the variable m based on the value of variables i and k and returns the modified value. The corresponding SPARC assembly program text file is shown in Fig. 2b. The content of specifications file, shown in Fig. 2c, states that the function main has no input parameters, has security signature Low and returns a low-security value. The global variables i, j, k, l and m have respectively the security levels Low, Low, High, Low and Low.

Figure 3 shows the window interface of SPARC PCC-SIF with the input file (Fig. 2b) and specifications file (Fig. 2c). The code window shows the program being verified and specification window shows the contents of specification file given by the user. The abstract execution window contains an abstract state for each instruction of the code. Each line the information that describes the execution state: The instruction id, the basic block, the security level of the program counter (security context), the label (if any), the op-code and the operands. Abstract state window shows the final security levels of registers, stack locations and memory locations.

IFD tree window shows the control dependence regions of each function in the program. IPD stack window shows the addresses of the branches of conditional jump instructions. Global offset window is for tracing the security levels of global variables and for tracing the pointers. To trace the order of the execution in step by step verification mode the user can use the execution traces window. The last window is verification conditions window, which shows the results of the verification process. Upon completing the verification process the content of the verification conditions window can be saved into ".elf" file and submitted to Twelf system.

The general information Panel gives some information about the code, the number of different conditions generated, the time elapsed for the verification process. The user can performs step-by-step verification process and can also traces the intermediate states.

Figure 4 shows the results of the verification of SPARC assembly program (Fig. 2b). Note that the results are intermediate and thus not sufficient enough to decide whether the program being verified is secure or not. In fact, the task of this step of verification is to collect the verification conditions that will be delivered to Twelf system to verify. Figure 5 shows the verification conditions to be submitted to Twelf theorem prover to generate their proofs.

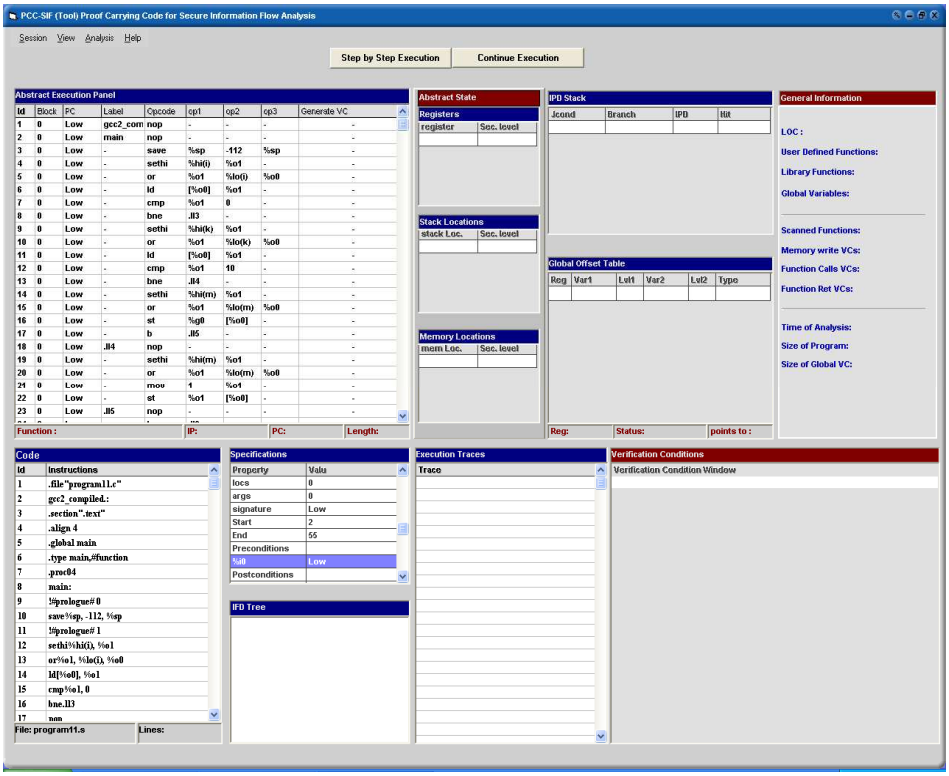

Fig. 3: User interface of the SPARC PCC-SIF tool

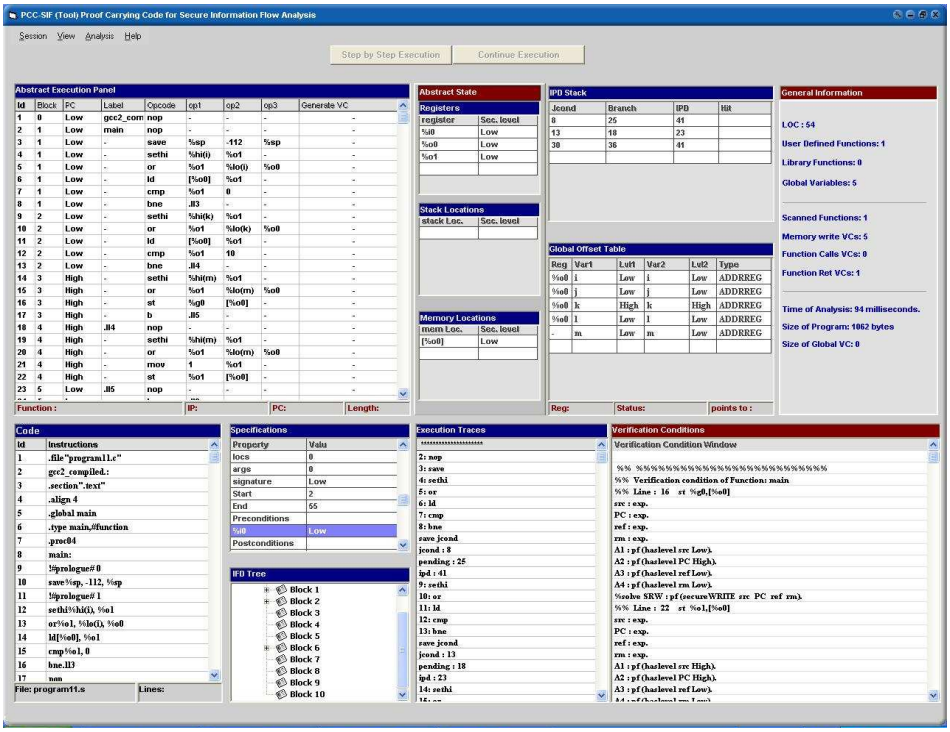

Fig. 4: Verification of SPARC assembly program of Fig. 2b

Table 2: Characteristics of test cases and number of security conditions

LOC	Program 1 52	Program 2 63	Program 3 76	Program 4 80	Program 5 139	Program 6 169	Program 7 172	Program 8 261	Program 9 428	Program 10 779
Verification condition generation	0.016	0.016	0.015	0.016	0.031	0.078	0.094	0.079	0.156	0.344
Proof generation and checking	0.030	0.025	0.016	0.032	0.015	0.024	0.045	0.026	0.420	0.630
Total (sec)	0.046	0.041	0.031	0.048	0.046	0.102	0.139	0.105	0.576	0.974

```
%% Verification condition of Function: main
%% Line : 16 st %g0,[%o0]
src : exp.
PC : exp.
ref : exp.
rm : exp.
A1 : pf (haslevel src Low).
A2 : pf (haslevel PC High).
A3 : pf (haslevel ref Low).
A4 : pf (haslevel rm Low).
%solve SRW :pf(secureWRITE src PC ref rm).
%% Line : 22 st %o1,[%o0]
src : exp.
PC : exp.
ref : exp.
rm : exp.
A1 : pf (haslevel src High).
A2 : pf (haslevel PC High).
A3 : pf (haslevel ref Low).
A4 : pf (haslevel rm Low).
%solve SRW :pf(secureWRITE src PC ref rm).
%% Line : 34 st %o1,[%o0]
src : exp.
PC : exp.
ref : exp.
rm : exp.
A1 : pf (haslevel src Low).
A2 : pf (haslevel PC Low).
A3 : pf (haslevel ref High).
A4 : pf (haslevel rm High).
%solve SRW :pf(secureWRITE src PC ref rm).
%% Line : 40 st %o1,[%o0]
src : exp.
PC : exp.
ref : exp.
rm : exp.
A1 : pf (haslevel src Low).
A2 : pf (haslevel PC Low).
A3 : pf (haslevel ref High).
A4 : pf (haslevel rm High)lve SRW : pf (secureWRITE
src PC ref rm).
%% Line : 45 st %g0,[%o0]
src : exp.
PC : exp.
ref : exp.
rm : exp.
A1 : pf (haslevel src Low).
A2 : pf (haslevel PC Low).
A3 : pf (haslevel ref Low).
A4 : pf (haslevel rm Low).
%solve SRW : pf (secureWRITE src PC ref rm).
%% Line : 52 ret
main : exp.
retValue : exp.
postcond : exp.
A1 : pf (haslevel retValue Low).
A2 : pf (haslevel postcond Low).
%solve SR:pf(secureRET main postcond retValue).
```

Fig. 5: Verification conditions file of SPARC assembly program file of Fig. 2b

The inputs of the Twelf tool are two files: an LF signature for reasoning about secure information flow in SPARC assembly programs and the file containing the verification conditions generated by SPARC PCC-SIF tool. Based on the signature given, the Twelf theorem prover attempts to verify the verification conditions and generates their detailed proofs.

```
src : exp.
PC : exp.
ref : exp.
rm : exp.
A1 : pf (haslevel src Low).
A2 : pf (haslevel PC High).
A3 : pf (haslevel ref Low).
A4 : pf (haslevel rm Low).
%solve
pf (secureWRITE src PC ref rm).
[Closing file example.elf]
example.elf:13.2-13.49 Error:
No solution to %solve found
%% ABORT %%
```

Fig. 6: Verification conditions file of SPARC assembly program file of Fig. 2b

tool are two files: An LF signature for reasoning about secure information flow in SPARC assembly programs and the file containing the verification conditions generated by SPARC PCC-SIF tool. Based on the signature given, the Twelf theorem prover attempts to verify the verification conditions and generates their detailed proofs.

Figure 6 shows the result of proving the verification conditions by Twelf theorem prover. We can see that the program is insecure as the theorem prover fails to generate its security proofs and aborts the theorem proving. By a visual inspection of the corresponding source program we can see that a low-security variable m has been assigned values in high-security region which may reveal information about the value of high-security variable k.

SPARC PCC-SIF tool is applied to few case studies. The programs that are used as test cases are written to reflect the desired program property for which the security tool verifies the assembly programs for (i.e., non-interference property). All case study programs are written in C language and compiled with GCC compiler (Version 2.95.3)[7].

Table 2 summarizes the time required to verify each program for secure information flow on a 2.4 GHz Intel machine with 1GB memory. The times include the time for scanning the program and producing verification condition for it, generating the proofs and checking the proofs. The time to verify these case studies ranges from 31 milliseconds to 1 sec. All proofs are obtained within fractions of a second.

For the case studies programs 3, 5 and 9 the time for generating and checking the proofs are also shown even though the theorem prover fails to generates proofs for them. The Twelf theorem prover aborts the theorem proving operation once it encounters a security predicate for which there is no proof derivation within the given logic program.

DISCUSSION

In absence of a reliable protection mechanism that can verify if a piece of downloaded code maintains confidentiality end-users may avoid executing the code due to the concern that it may leak confidential data. While it protects end-users, this strategy; however prevents them to benefit from the richness of the web.

There are two ideas for utilizing secure information flow analysis for protecting data confidentiality.

Idea 1: Developing Secure Software. In this idea secure information flow analysis is used to help in detecting unauthorized information flows when writing the program and thus helps in developing secure software. It could be used as a static analysis tool that spots the potential leakage and alerts the programmer who will respond by rewriting the program in such that it obeys the information flow policy. Here, secure information flow analysis can be carried out on source code.

Idea 2: Preventing Malicious Programs. The idea here is to use the secure information flow analysis as an analyzer to verify the security of untrusted mobile programs. A piece of untrusted code is analyzed with the goal of establishing its security. If the mobile program has been proved secure, then it is allowed to execute, otherwise its execution is stopped. Here, secure information flow analysis is carried out on the machine code (assembly code). Our research work subsumes under this category. It is obvious that analyzing machine code is much more difficult than analyzing the source code.

Issues with information flow checking of assembly programs: To perform information flow analysis of assembly programs, the issues we face include: (1) Reuse of registers: a register holds values of different variables at different program points; registers cannot be assigned fixed security levels; (2) Assembly programs lack the high-level control flow structures, which may present in source programs and are necessary for tracking implicit information flows; this calls for a mechanism to retrieve such structures; (3) Memory aliasing between memory locations cannot be easily reason about. Memory aliasing refers to the situation in which two pointers point to same memory location.

Implications: An important feature of the proposed tool is that it certifies assembly programs generated by general-purpose off-the-shelf compilers. This gives the code producers flexibility in the choice of the high-level language, in which programs are written and allows end-users to check a wide variety of programs and thus the tool can be used effectively for protecting confidentiality. As our technique checks only the output of the compiler, it eliminates the dependence of security checking results on the correctness of the compiler. In addition, it leads to separating of security policy from the source language, which in turn, enables the code consumer to extend the security properties against which the untrusted code is checked.

CONCLUSION

We have presented a security technique that allows the verification of assembly programs for secure information flow. The tool SPARC PCC-SIF was developed and is based on the concept of proof-carrying code and provides an automatic verification of SPARC assembly programs for secure information flow. To perform the secure information flow verification it is required that all variables in the program are associated with security levels. The presence of explicit proofs that are generated and checked provides distrustful users with a strong guarantee of protecting confidentiality making them more confident in the security technique and the tool.

REFERENCES

1. Goguen, J.A. and J. Meseguer, 1982. Security policies and security models. Proceedings of IEEE Symposium on Security and Privacy, (SCP' 82), IEEE Computer Society, USA., pp: 11-20. http://doi.ieeecomputersociety.org/10.1109/SP.1982.10014
2. Denning, D.E. and P.J. Denning, 1977. Certification of programs for secure information flow. Commun. ACM., 20: 504-513. http://doi.acm.org/10.1145/359636.359712
3. Sabelfeld, A. and A. Myers, 2003. Language-based information-flow security. IEEE J. Selected Areas Commun., 21: 5-19. DOI: 10.1109/JSAC.2002.806121

4. Necula, G., 1997. Proof-carrying code. Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Jan. 15-17, Paris, France, pp: 106-119. http://doi.acm.org/10.1145/263699.263712
5. Muthana, A., A.A. Ghani, M. Ramlan and H. Selamat, 2007. Information flow type system for proof carrying code. Int. J. Comput. Sci. Network Secur., 7: 177-184. http://paper.ijcsns.org/07_book/200707/20070724.pdf
6. Zdancewic, S. and A. Myers, 2002. Secure information flow via linear continuations. Higher-Order Symbol. Comput., 15: 209-234. http://www.cis.upenn.edu/~stevez/papers/ZM02.pdf
7. Woehr, J., 1994. What's GNU? Embed. Syst. Program., 7: 70-72, 74.
8. Bonelli, E., A. Compagnoni and R.Medel, 2004. SIFTAL: A typed assembly language for secure information flow analysis. Technical report, Stevens Institute of Technology, Hoboken, NJ. http://www.cs.stevens.edu/~abc/publications/siftal Long.ps
9. Medel, R., A. Compagnoni and E. Bonelli, 2005. Non-interference for a typed assembly language. Proceedings of 2005 Workshop on Foundations of Computer Security, May 13-13, Chicago, IL., pp: 1-12. http://www.cs.stevens.edu/~abc/publications/sifon. pdf
10. Yu, D. and N. Islam, 2006. A typed assembly language for confidentiality. Lecture Notes Comput. Sci., 3924: 162-179. DOI: 10.1007/11693024_12
11. Barthe, G., A. Basu and T. Rezk, 2004. Security types preserving compilation. Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation, Jan. 11-13, Venice, Italy, pp: 2-15. http://www.msr-inria.inria.fr/~rezk/publication/Barthe-Basu-Rezk.php
12. Barthe, G. and T. Rezk, 2005. Non-interference for a JVM-like language. Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Jan. 10-10, ACM Press, New York, USA., pp: 103-112. http://doi.acm.org/10.1145/1040294.1040304
13. Genaim, S. and F. Spoto, 2005. Information flow analysis for java bytecode. Proceeding of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, Feb. 4-4, Springer, Paris, France, pp: 346-362. DOI: 10.1007/b105073
14. Bieber, P., J. Cazin, V. Wiels, G. Zanon, P. Girard and J.L. Lanet, 2002. Checking secure interactions of smart card applets: Extended version. J. Comput. Secur., 10: 369-398. http://iospress.metapress.com/content/vb59hrkba34 d8nm0/
15. Bernardeschi, C. and N. Francesco, 2002. Combining abstract interpretation and model checking for analysing security properties of Java bytecode. Proceedings of Verification, Model Checking and Abstract Interpretation, Jan. 21-22, Venice, Italy, pp: 1-15. http://portal.acm.org/citation.cfm?id=646541.6961 78&coll=GUIDE&dl=
16. Avvenuti, M., C. Bernardeschi and F. Francesco, 2003. Java bytecode verification for secure information flow. ACM SIGPLAN Notic., 38: 20-27. http://doi.acm.org/10.1145/966051.966055
17. Bernardeschi, C., N. De Francesco and G. Lettieri, 2002. An abstract semantics tool for secure information flow of stack-based assembly programs. Microprocess. Microsyst., 26: 391-398. DOI: 10.1016/S0141-9331(02)00064-9
18. Pfenning, F. and C. Schurmann, 1999. System description: Twelf: A meta-logical framework for deductive systems. Proceedings of the 16th International Conference on Automated Deduction, Jan. 7-10, Springer, Berlin, pp: 202-206. http://portal.acm.org/citation.cfm?id=648235.753634
19. Ball, T., 1993. What's in a region? Or computing control dependence regions in near-linear time for reducible control flow. ACM Lett. Program. Languages Syst., 2: 1-16. http://doi.acm.org/10.1145/176454.176456
20. Harper, R., F. Honsell and G. Plotkin, 1993. A framework for defining logics. J. Assoc. Comput. Mach., 40: 143-184. http://doi.acm.org/10.1145/138027.138060