I.J. Wireless and Microwave Technologies, 2012, 3, 34-40
Published Online June 2012 in MECS (http://www.mecs-press.net)
DOI: 10.5815//ijwmt.2012.03.06

Available online at http://www.mecs-press.net/ijwmt

A Data Flow Behavior Constraints Model for Branch Decision-making Variables

Lu Yan, Wang Dan, Zhao Wen Bing

College of Computer Science, Beijing University of Technology, Beijing, china

Abstract

In order to detect the attacks to decision-making variable, this paper presents a data flow behavior constraint model for branch decision-making variables. Our model is expanded from the common control flow model, it emphasizes on the analysis and verification about the data flow for decision-making variables, so that to ensure the branch statement can execute correctly and can also detect the attack to branch decision-making variable easily. The constraints of our model include the collection of variables, the statements that the decision-making variables are dependent on and the data flow constraint with the use-def relation of these variables. Our experimental results indicate that it is effective in detecting the attacks to branch decision-making variables as well as the attacks to control-data.

Index Terms: Program behavior; branch decision-making variable; control flow; dependence relation

1. Introduction

According to the TCG specification, the trust of an entity is based on its behavior's trust. Software's trust can be measured by its behavior when it is executing [1]. Taking trust measurement to program dynamically is hard. To date, there have been many related researches focusing on how to describe the software's behavior. These researches mainly established of software control flow behavior models based on control flow model through the static analysis or dynamic learning [2][3]. There are two kinds of control flow behavior models at present including system call-based model and function call-based model. Control-flow behavior model has been proven that it can effectively detect common attack types, such as malicious code injection attacks and return-to-libc attacks, .etc.

The behavioral models based on the traditional control-flow can effectively detect the attacks of the control-data. Nevertheless, since the control-flow behavior model only concerns about the integrity of control-data, it doesn't inspect the data flow information of program, therefore, when an attacker tampers with other non-control-data, it may change the software control flow without corrupting the control-data, hence, the traditional

This work was supported by a grant from the Major State Basic Research Development Program of China (973 Program) (No.2007CB311106)

control-flow behavior model can not detect such kind of attack which is relatively a common sort of threat [4]. Among them, the control-data refers to control flow related data of the software, for example, return addresses of function call, function pointers, etc.. On the other hand, non-control-data includes configuration data, user data, decision-making data (the variables referenced by the branch statement) and so on.

This paper proposes a data flow behavior constraint model for branch decision-making variables and it can be used to verify program's behavior while running. Our model made some improvements for the common control-flow behavior model by mainly increased the study on the relationship of data flow of decision-making variable. The contribution of this paper includes: (1) our model can detect the branch statements related behavior which depends on the values of decision-making variables; (2) our model can extract constraint of data flow behavior which is relevant with decision-making variables of branch statements; (3) our model can detect whether it meet the constraints of data flow as well as verifying the relation of function calls during running time, so it can guarantee the correctness of the decision-making variables and authenticate the behavior of branch statements. When an error happens during the function call or an abnormal data flow occurs, the system will give an alarm.

2. Branch-Reserving Call Graph

Generally speaking, a software behavior model should be composed of two parts including the function calls relations and branch statement of decision-making variables for the constraints of data flow behavior. The former is not the focus of this paper, we mainly illustrate the latter.

2.1. Consructing BRCG

The function call is regarded as the behavior granularity for control flow behavior in this paper, therefore we only take into consideration the branch statements which have control dependence relations with function call statements and function return statements. The control dependence means if the execution of the statement *S1* is decided by the execution state of the statement *S2*, then we say *S1* depends on *S2*.

In order to extract branch statements that the function calls depend on, we adopt procedure structure model which is proposed in [5]. It added the information of branch statements into the function call graph and formed a graph called Branch-Reserving Call Graph, denoted by BRCG. The brief definition of the BRCG is described as follows:

BRCG is defined as a triple <N, S, B>, among them,

1) N is a set, its element includes functions-call statements and branch-call statements.
2) S is a set of sequential relations, where for $\forall <n_1, n_2> \in S, n_1 \in N$ and $n_2 \in N$.
3) B is a set of branching relations, where for $\forall <n_1, n_2> \in B, n_1 \in N$ and $n_2 \in N$.
4) for $\forall <n_1, n_2> \in S$ and $\forall n_3 \in N$, $<n_1, n_3> \notin B$ and
   for $\forall <n_1, n_2> \in B$ and $\forall n_3 \in N$, $<n_1, n_3> \notin S$.

Generally, statements including *if* statements, *case* statements and *loop* statements in program are all regarded as branching relations. The sub-graphs of BRCG for each function are linked through the function call point. In this way, we can get a complete Branch-Reserving Call Graph. For example, the BRCG of the function *foo* is shown in the Fig. 1.

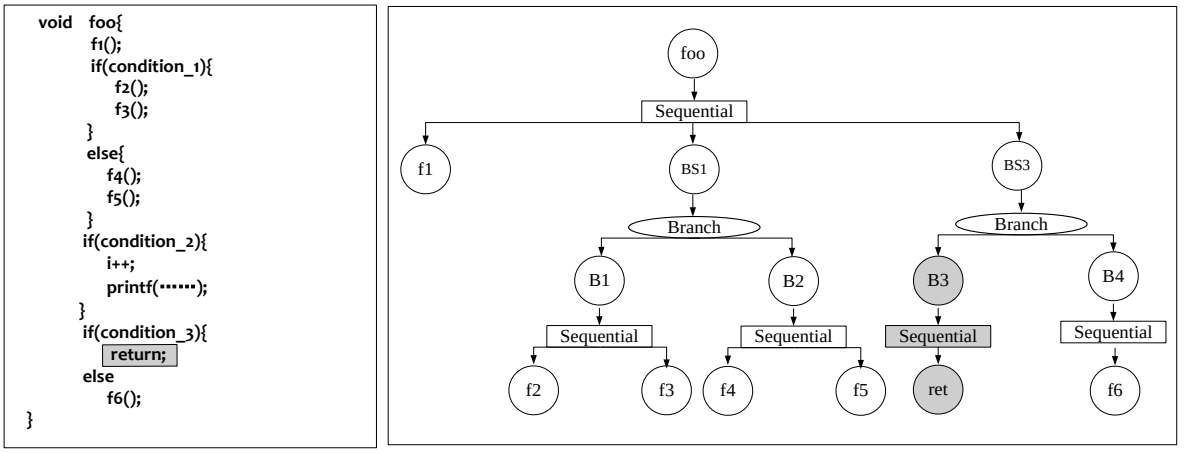

Fig 1. An example of BRCG

Due to the function call and function return are two important control flow behaviors, it is necessary to add the function returns related information into the definition of BRCG. In this way, BRCG can describe the branch statements which function return would depend on, as shown in the Fig. 1(the gray part).

N in the BRCG is defined as follows: N is a set, its elements are composed of function call statement, branch statements and function return statements. The others use the definition of [5].

2.2. Pruning BRCG

Because there may be lots of information about branch statements in BRCG, it will take large overheads to create verification model on the basis of BRCG. Therefore, it is necessary to prune appropriately for BRCG. For minimizing false negative, this paper takes the following conservative strategy: prune the sub-graphs of basic function only.

For the well-designed software, each component plays a different role in the system. On the basic level, functions only finish the low-level and relatively simple functions, such as to control equipments running, and achieve basic operations of data structures such as stack, queue .etc as well as achieve basic system functions such as writing log. Reference [6] named this kind of function utility routines. Functions, which are on the higher-level, take advantage of this mechanism provided by basic functions to achieve complicated system functions with certain strategy.

The purpose of tampering with the decision-making variables of branch is to bypass security checking or logic checking so as to acquire functions of software. Utility routines are merely achieved details of low-level, they do not involve decision and judgment of logic, business or security of higher-level. Therefore, the branch actions of utility routines could be not involved in the scope of dynamic verification. Based on that, our strategy of pruning is to prune the sub-graphs of utility routines in BRCG.

Reference [6] indicates us an important feature that any utility routines called many times in the different position of software code may have higher *fan-in* value, so we can identify utility routines by calculating their *fan-in*. We adopt the following calculation given in [6]:

1) S is the set of all functions defined in software.;S={f_1,f_2,……f_i,…,f_n}

2) IN_{f_i} is the count that function f_i being called, that is the *fan-in* value of function f_i.

3) $M = \frac{\sum_{i=1}^{n} IN_{f_i}}{|S|}$

M is the threshold, which is the average of *fan-in* of all functions. For any $f_i \in \{f_1, f_2 ..... f_i ... f_n\}$, if $IN_{fi} > M$, then we can determine that f_i is a utility routine. Assuming that the relations of function call are shown in Fig. 2, every function's *fan-in* and *M* value showed in Table 1. From Table 1, we can learn that the *fan-in* of function *f6* and *f7* are greater than M, so they can be considered the utility routines.

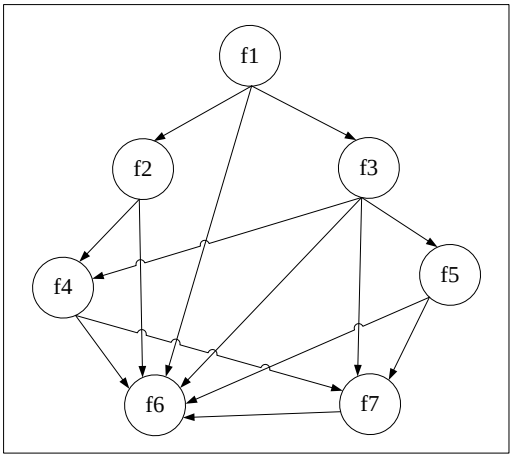

Fig 2. An example of function call graph

Table 1 Function and Its Fan-in

function	Value of *fan-in*
f1	0
f2	1
f3	1
f4	2
f5	1
f6	6
f7	3
M = 2	

3. Behavior Constraint of Data Flow Associated With the Branch Decision-Making Variables

Since the behavior of branch statements depends on the value of decision-making variables and attackers may change the behavior of branch statements by tampering the value of decision-making variables, if the decision-making variables can be found firstly and make appropriate protection for them, the behavior of branch statements may be verified and the violation may be forbidden.

Generally, the corruption of variables occurs when a statement illegitimately modifies the data in memory address which is out of its writable ranges. According to the definition given in [7], supposing valuable v is

defined in statement *dst* and referenced in statement *ust*, moreover, v is not redefined between *dst* and *ust*, then *ust* is data flow dependent on *dst*, this is called *use-def* relation and denoted by $v(ust,dst)$. Therefore, detecting whether data flow during running time is conformed with *use-def* relation or not can dynamic detect the occurence about corruption of the decision-making variables.

On the other hand, because there is dependence relation between variables, only detecting the decision-making variables is insufficient. As for *c=a+b*, supposing *c* is a decision-making variables, then *c* would be indirectly influenced by tampering *a* or *b*, thus, it is necessary to detect variables which are directly or indirectly depended by decision-making variables. Using program slicing techniques could extract variables and statement set which are depended by decision-making variables. Program slicing set is a subset of program statements, contains statements depended by variables at a program point directly or indirectly [8], the dependence includes control dependence and data dependence. In this paper, the control dependence will be omitted, we only extract data dependence statement, then further extract variables depended by decision-making variables.

For the branch reserved calling graph of a software G<N,S,B>, G′<N′,S′,B′> is its corresponding graph after pruning. Supposing the set of all branch statement in N' is $Br=\{br_1,br_2...br_i...\}$, for $br_i \in Br$, decision-making variable set of br_i is denoted as V_i.

Supposing for decision-making variable $var \in V_i$, we can calculate the statement set which var is dependent on at br_i by using program slicing technique, it is denoted as $slice_{(br_i,var)}=\{s_1,s_2,....s_i,...\}$. For statement $s_i \in slice_{(br_i,var)}$, the variable set referenced by s_i is denoted by $ref(s_i)$, the variables set modified by s_i is denoted by $def(s_i)$. According to the program slicing algorithm, for all statements s in $slice_{(br_i,var)}$, $ref(s)$ is the set of variable depended directly or indirectly by decision-making variable var at statement s.

Supposing $ref(s_i)=\{v_j,v_k...v_n\}$ for any $s_i \in slice_{(br_i,var)}$, according to the program slicing algorithm, for any $v_l \in ref(s_i)$, there is a statement set $dstm_{(v_l,s_i)} \subset slice_{(br_i,var)}$ and for all statements s which belong to $dstm_{(v_l,s_i)}$, $v_l \in def(s)$ and v_l is not redefined between s and s_i, this *use-def* relation is denoted by $v_l(s_i,s)$.

The *use-def* relation above is defined as the behavior constraint of data flow. When a process is running, data flow should conform to its behavior constraint. If variable v conforms to *use-def* relation for a statement s, it means when v is referenced at statement s and the statement which modifies v most recently is in the set $dstm_{(v,s)}$, then v can be considered trusted at statement s, otherwise it is untrusted.

When the branch statement br_i is running, if all decision-making variables in the set V_i are trusted and the variables that br_i's decision-making variables are dependent on are trusted at corresponding statement in the current execution path respectively, then br_i's behavior is trusted.

During running time, dynamic instrumentation technology can be used to achieve the dynamic verification. The detail is omitted since space is limited.

4. Security Analysis of this Model

The authentication function of SSH server is used to illustrate the effectiveness of our model. Ignoring some detailed parts, some piece of SSH code is shown in Fig. 3.

```
void do_authentication(char *user, ...){
1: int authenticated = 0;
   ...
2: for(;;) {
  /* Get a packet from the client */
3: type = packet_read();
  //calls detect_attack() internally
4: switch (type) {
     ...
5:   case SSH_CMSG_AUTH_PASSWORD:
6:     if (auth_password(user, password))
7:         authenticated =1;
     case ...
  }
8: if (authenticated)
9:         return;
  }
}
```

Fig 3. some code of do_authentication()

SSH server certifies connection by function do_authentication(). This function can certify remote login user by infinite loop of multiple user authentication mechanisms. If remote users could pass one of mechanisms, the certification succeeds. Local variable *authenticated* is used for marking whether the certification passes authentication or not. This function receives user input further in statement 3th and calls function detect_attack(). Since function detect_attack() exists integer overflow vulnerability and the vulnerability could be triggered as soon as a set of data that is specifically designed is inputted on remote node, so that the remote users can modify any data of memory address easily to revalue variable *authenticated* to *1* to break the loop, and bypass the authentication mechanism.

The *fan-in* value of function do_authentication() is *1*,and statement 9th is its return statement, so both statement 8th and 9th should be included in BRCG that have been pruned. The data flow constraints of decision-making variable of branch statement 8th would be into the range of verification. If variable *authenticated* is modified during detect_attack() running, as *authenticated* variable does not belong to the set of writable variables, the trusted status of variable *authenticated* would be set to *untrusted*. While branch statement 8th is being executed, we are able to check the variables referenced by the statement 8th to discover abnormal actions of a process.

5. Concluding Remarks

We described a new model for detecting software problems at run time by analyzing and verifying the data flow for decision-making variables. Our model can be effective in detecting the attacks to branch decision-making variables as well as the attacks to control-data. While our work is still at an early stage and it incurs much overhead, we intend to expand and optimize our method to include the analysis of additional parameters and rules that define their relations. In our future works, we will further enhance the practicality of the expected behavior model.

References

[1] Trusted Computing Group, http://www.trustcomputing,org
[2] M.Abadi, M.Budiu, Ú.Erlingsson, J.Ligatti, "Control-flow integrity principles, implementations, and applications" ACM Transactions on Information and System Security, 2009, 13(1), pp.1-40.
[3] H.Feng, O.Kolesnikov, P.Fogla, W.Lee, W.Gong, "Anomaly DetectionUsing Call Stack Information" In IEEE Symposium on Security and Privacy, Oakland, California, 2003, pp.62-76.
[4] S.Chen, J.Xu, E.C.Sezer, P.Gauriar, R.K.Iyer, "Non-control-data attacks are realistic threats" in Proceedings of 14th USENIX Security Symposium, Berkeley, CA, USA, 2005, pp.12-16
[5] T.Qin, L.Zhang, Z.Zhou, D.Hao, J.Sun, "Discovering use cases from source code using the branch-reserving call graph" In Proceedings of the 10th Asia-Pacific Software Engineering Conference. IEEE Computer Society,Washington, D.C, 2003, pp.60–67.
[6] H.Lhadj, A.Braun, D.Amyot, T.Lethbridge, "Recovering Behavioral Design Models from Execution Traces" Software Maintenance and Reengineering, 2005, pp.112-121.
[7] A.Aho, R.Sethi, J.Ullman. Compilers,Principles, Techniques, and Tools, Addison-Wesley Publishing Company, Massachusetts, 1986.
[8] S.Horwitz, T.Reps, D.Binkley, "Interprocedural slicing using dependence graphs" ACM SIGPLAN Notices, 2004, 39(4), pp.229-243