Using Branch Correlation to Identify Infeasible Paths for Anomaly Detection

Xiaotong Zhuang Tao Zhang Santosh Pande

IBM T.J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598
{xzhuang, taozhang}@us.ibm.com

Georgia Institute of Technology
College of Computing, 801 Atlantic Drive
Atlanta, GA, 30332-0280
santosh@cc.gatech.edu

Abstract

In this paper, we propose a system called Infeasible Path Detection System (IPDS) to combat memory tampering attacks causing invalid program control flows. In our system, the compiler analyzes correlations between branches and then the analyzed information is conveyed to the runtime system. The runtime system detects dynamic infeasible program paths by combining compiler determined information with runtime information to check the legality of the path taken during execution. IPDS achieves a zero false positive rate and can detect a high percentage of memory tampering for many attacks in which the tampering actually causes a change in control flow. Moreover, IPDS only incurs a modest amount of hardware resource and negligible performance penalty.

1. Introduction

Modern computers are prone to security vulnerabilities due to various program bugs, mis-configurations, improper uses, etc. Such vulnerabilities could lead to attacks on the computer systems. The most commonly exploited vulnerabilities are buffer overflows (BOA) [1], which exploit the fact that bounds check is missing in programming languages such as C. In traditional buffer overflow attacks, by sending an unexpected long string to the program, the attacker is able to overwrite the function return address and redirect the program control to the code injected by him. There has been a wealth of research work [2][3][4][5][6] on how to prevent return addresses/ function pointers from being tampered and malicious code being injected and executed. Code injection attacks are expected to have less impact in the future since the countermeasures are extensively studied, implemented and deployed. However, there is another category of attacks in which the attacker does not inject his code into the attacked program and execute it. In these types of attacks, the attacker still relies on memory tampering by buffer overflows etc., but other critical program data like user ID, user inputs or decision-making data are tampered. These types of attacks are much more difficult to detect, since the program control is only transferred inside the original program and is never transferred into the attacker's hands or the injected code.

Anomaly detection systems [7][8][9][10][12][13] have been proposed as a general solution to detect attacks and enhance system security. Most early anomaly detection approaches analyze audit records against profiles of normal user behavior. Forrest et al. [7] discovered that a system call trace is a good way to depict a program's normal behavior, and anomalous program execution tends to produce distinguishable system call traces. A number of follow-up work, such as [12][13], focus on how to represent system call sequences compactly with finite-state automata (FSA). Recent advances [8][9][10] propose to incorporate more program information than system call traces to achieve a stronger detection capability. However, incorporating more information and monitoring at a fine granularity could lead to a high false positive rate.

In this paper, a novel anomaly detection technique called Infeasible Path Detection System (IPDS) is proposed. We systematically explore ways of utilizing compiler analysis to construct correlations among branches. The compiler gathered information is then made available to the runtime system and used to detect infeasible dynamic program paths caused by attacks. Our system works with low runtime overhead and a zero false positive rate. Experimental results indicate that the scheme is able to detect a wide range of memory tampering based attacks.

This work was partially supported by NSF grants CyberTrust CNS 0524651 and CCF 0541273.

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

The rest of the paper is organized as follows: Section 2 gives background knowledge and motivation; Section 3 talks about our machine model and assumptions; Section 4 discusses branch correlations; Section 5 elaborates our approaches; Section 6 presents experiments and results; Section 7 discusses related work and finally section 8 concludes the paper.

2. Background and Motivation

There has been a lot of research [2][3][4][5][6] on preventing malicious code injection attacks, in which the attacker tampers function return addresses or function pointers by exploiting vulnerabilities such as buffer overflows. Code injection attacks are classical and have been well studied. However, it is important to note that by exploiting buffer overflow kind of vulnerabilities, the attacker does not necessarily inject his own code to launch an attack. Figure 1 gives an example.

The code first verifies the user, typically through user inputs such as passwords, etc. The result is assigned to the variable "user". After user verification, there are a number of operations that are performed depending on the user's identity. The code frequently compares the variable user with "admin" to determine whether certain operations should be allowed. Note that the two "if statements" should always take the same direction at runtime, i.e. either they are both taken or both not-taken. However, in between of the two if statements, the program interacts with the user again to get some other inputs. Unfortunately, there is a buffer overflow vulnerability. Input "someinput" is actually manipulated by the attacker, which craftily modifies the variable user through a buffer overflow attack. It is clear that throughout the attack, no malicious code is injected and executed, however the attacker can still get privilege escalation and other malicious actions might be taken after the program now assumes the user is the administrator. Chen et al. made the same observation in [20].

```
1.    char str[SIZE], user[SIZE];
2.
3.    verify_user(user)
4.    if (strncmp (user, "admin", 5)) {
5.      ...
6.    } else {
7.      ...
8.    }
9.    strcpy (str, someinput);
10.   if (strncmp (user, "admin", 5)) {
11.     //superuser privilege
12.     ...
13.   } else {
14.     ...
15.   }
```

Figure 1. An attack without code injection (cited from [10]).

In reality, such vulnerabilities/bugs are frequently seen. Also, buffer overflow is only one type of vulnerabilities. New vulnerabilities are being discovered almost every day. Recognizing the difficulty to identify all potential vulnerabilities of a program, researchers have proposed anomaly detection as a general means to detect incidence of an attack [7][8][9][10][12][13]. An anomaly detection system monitors the program execution and raises an alarm if there is unusual/abnormal program behavior.

It has been well established that monitoring program behavior dynamically is a good way to detect anomalies induced by attacks that exploit software vulnerabilities. Such program behavior could include system calls, function calls, control and data flows etc. Our basic assumption is that the data should remain the same when it is fetched back from the memory after the previous access. The compiler derives which paths are feasible/infeasible based on this assumption. We give a simple example in Figure 2. In Figure 2, if the path goes from BB1, BB2 to BB4, then the backward branch must be taken at the end of BB4 looping back to BB1 because we know x<0 at the end of BB1, so it must be less than 10 as well. If we see a path going from BB1, BB2 to BB4, but not from BB4 to BB1, variable x must be corrupted when it is loaded back from the memory. Also, the execution must go to BB2 in the second iteration, since we know x is not changed, therefore the branch at the end of BB1 should take the same direction.

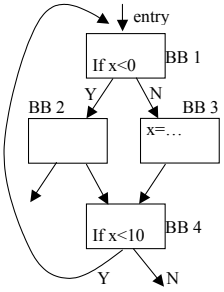


Figure 2. An infeasible path caused by memory tampering.

3. Attack Model and Machine Model

There are two attack models to which our scheme can be applied. In the first model, the attacker can only interact with the system through a number of input channels, such as keyboards, network connections etc. He can launch intrusion activities through program bugs such as buffer overflows, format string attacks, integer overflow etc. In the second model, we assume that there are multiple processes running on the same system, and some of them are corrupted and malicious. The malicious process may snoop/tamper with other processes' memory space through shared memory or undue privileges, or it can tamper with other processes' critical data through malicious inputs just as in the first model. The second model happens when the

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

attacker has gained partial control of the system and intends to conduct further malicious activities.

We have the following assumptions in our machine model: 1) we introduce some hardware components for security inside the processor; 2) process context is protected and is considered secure during context switches, including registers and the anomaly detection state associated with each protected process; 3) read-only memory can be enforced properly by the processor, thus program code and memory resident constants defined statically can be considered secure. On the other hand, any non-constant data residing in the external memory is considered non-secure; 4) Code accessing the critical data are available for compiler analysis. Our scheme is conservative when pointers are passed to library code without source.

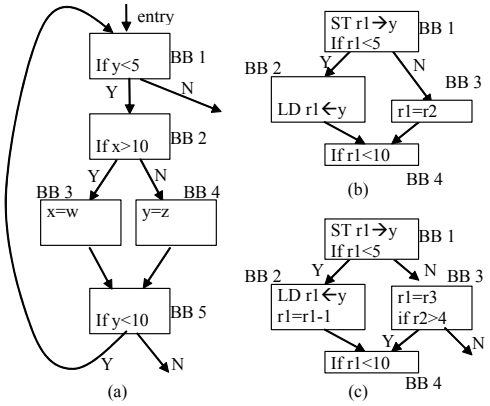


Figure 3. An example of branch correlation.

The above assumptions are realistic. Recently, there is a strong drive to introduce secure processors into high-end and desktop computing domains [21][22][23]. In particular, Intel plans to incorporate security support to its processors within the next two to three years [23]. Thus, secure desktop/server processors could become available in the near future. Also, process context is very basic and critical information for a process and should be protected properly in any reasonable secure systems. Read-only data can be considered to be secure since it is simple for the hardware to enforce read-only property. Finally, most attacks, such as buffer overflow attacks, break into the system through tampering to the external memory. Normally they cannot garble values in registers directly. However, register contents could be indirectly tampered by a load of non-secure memory resident data. Thus, in our scheme, we focus on *non-constant memory resident data values*, which are the targets of attacks. Our goal is to detect infeasible program paths due to the tampering to those data values.

4. Branch Correlations

Each path consists of a number of branches and thus, branches provide the basic unit in path verification. In this section, we illustrate how branches affect each other with examples. It is well known that branches are not totally independent of each other but could be correlated. There are multiple ways how branches affect each other. Branch correlations have been extensively studied previously such as in [16][17]. We give an example in Figure 3.

In Figure 3.a, we assume all variables are memory resident and are not constants. The figure shows several interesting cases. If the branch in BB1 is taken and the path follows BB1→BB2→BB3, then BB5 should be taken as well, because y must be less than 5 if BB1 is taken, and along the path, y is not modified. Thus, the condition y<10 must be true. However, if the path follows BB1→BB2→BB4, then the direction of BB5 cannot be determined statically, since y is assigned a new value and normally we do not know whether this value is smaller than 10 or not. For variable x, it is used in BB2 in the conditional branch and is changed in BB3. Suppose the branch in BB2 is not taken for the first time, when the execution goes back to BB2 in the next loop iteration, we know this branch will not be taken, since variable x is loaded again without being changed from the last definition. If x is tampered in memory, it could lead to a different branch outcome and we should be able to detect that. However, if BB2 is taken in one iteration, x will be assigned a new value, which causes the branch outcome in BB2 to be unknown since normally we do not know whether the new value of x is larger than 10 or not.

From the above example, we can identify three scenarios in which branches affect each other. 1) The variables involved in a conditional branch are redefined somewhere before the branch is reached again. In such cases the outcome of the branch will normally become unknown. 2) The variables involved in the conditional branch are not redefined anywhere before the branch is reached again. In such cases the outcome of the branch should be the same as the last outcome. 3) A branch's condition subsumes another branch's. Here "subsume" is used to indicate that if a variable in one range, then it must be in the other range, e.g., range [0, 5] subsumes range [0, 10]. The branch in BB1 subsumes the branch in BB5, because range y<5 subsumes range y<10. Note that scenario 2 can be regarded as a special case of scenario 3, in which one dynamic instance of a branch subsumes another instance (their ranges are same).

Figure 3.b shows how a piece of real code may look like as another example. In BB1, variable y is

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

stored in memory. If the branch in BB1 is taken, we know that the value of y is less than 5. We also know that the load of y in BB2 should get a value less than 5 and the branch in BB4 should be taken. However, if the dynamic program path is BB1→BB3→BB4, r1 is redefined in BB3 and generally we do not have information for the new value of r1. Thus, to be conservative, we have to regard both taken and not-taken outcomes of the branch in BB4 as possible.

Finally, Figure 3.c shows that sometimes we are able to analyze more complicated cases. In Figure 3.b, after r1 is reloaded in BB2, there is no further redefinitions to r1. However, in some cases, even after a variable is redefined, we can still have certain information about its new value, which is especially true if the new definition is done through simple arithmetic operations. In this example, r1 is decreased by 1. After r1 is decreased, we still know that the branch in BB4 must be taken because y is known to be less than 5; after it is loaded and decreased, it should be still less than 5 and thus less than 10 as well.

5. Our Approach

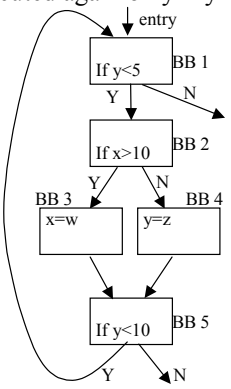
This section elaborates our infeasible path detection approach.

5.1 Branch Status Vector and Branch Action Table

The first task is to design proper data structures for the infeasible path detection scheme. In our scheme, we need to record the expected direction for each branch (obtained by combining both static information collected by the compiler and the runtime information), so that at runtime after the branch is executed and the actual direction is known, we can compare the actual direction with the expected direction detect anomalies. We introduce *Branch Status Vector (BSV)* for that purpose. Also, from the discussion of branch correlations, it is clear that whether a branch correlation exists at runtime heavily depends on whether certain variables get redefined and how they are redefined, which in turn depends on which dynamic path the program takes or the directions of dynamic branches. Thus, we need another data structure to record the interactions (correlations) between branches. We introduce *Branch Action Table (BAT)* for that purpose.

For each conditional branch, we need to keep two bits to encode three possibilities, namely taken, not-taken and unknown. At runtime, after a branch instruction is executed, we first verify its actual direction with the expected direction. If they do not match, an infeasible path has been detected. If no mismatch is found, we update the branch status vectors properly according to the actual direction of the branch and the branch action table.

An example is shown in Figure 4. In the example, three branches, located at the end of BB1, BB2 and BB5 are present. To facilitate our discussion, we name them BR1, BR2 and BR5 respectively. Their branch status vectors are tracked on the right side of the figure. Initially, all branch statuses are UN, i.e. unknown. Assume the first dynamic instance of BR1 is taken. After the execution of the branch, we first verify the branch. Since "unknown" matches any direction thus no mismatch is found. Then we update the status vectors. The status vectors of BR1 and BR5 need to be updated since BR1 subsumes BR5 and of course BR1 subsumes itself. So we set status vectors of BR1 and BR5 to "taken" as the current expected direction. Those expected directions are obviously optimistic. For example, BR1 will take the same direction when it is executed again only if y is not redefined in between.



Path	BR1	BR2	BR5
Init	UN	UN	UN
BB1	T	UN	T
BB2	T	UN	T
BB3	T	UN	T
BB5	T	UN	T
BB1	T	UN	T
BB2	T	NT	UN
BB4	UN	NT	UN
BB5	UN	NT	NT

Figure 4. An example on how to update the BSV.

Let us go back to the example. The program path goes to BB2 and BR2 is taken at runtime. The status vector of BR2 is then set to "unknown" instead of "taken". The reason is that entering BB3 causes variable x to be redefined to an unknown value, which means, the direction of BB2 should become unknown now. Then the program path goes into BB5 and BR5 is taken. For the second execution of BR1, since there is no redefinition to its depending variable y, the branch is taken again and this matches the status vector. If BR1 is not taken at runtime, then an attack is detected. When BR2 is executed again, the status vector shows its direction to be unknown, therefore any direction is correct. This time, it goes to BB4, i.e. not-taken. This causes the status vector of BR5 to be unknown, since variable y is redefined to some unknown value. The verification of BB5 is again correct since the status vector gives "unknown".

During the discussion of the above example, we discussed various actions to update status vectors after a dynamic branch is executed and its direction is known. Those actions encode correlations between branches. How to construct and store those actions is

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

the major job of the compiler. The actions are stored in a data structure called *Branch Action Table (BAT)*. It records which branches' branch status vectors should be updated and how to update them after a dynamic branch is executed.

In the simplest form (without concerns to the table size), we can store the BAT as a two dimensional array with br_num*br_num elements. Here br_num is the number of branches (we temporarily assume all branches are recorded, later we will talk about how to reduce that number).

```
INPUT:   Program code in a function (CFG form),
         br_num: Num of Branches
OUTPUT:  BAT: The Branch Action Table
         BCV: Branch Check Vector

enum br_action { SET_T, SET_NT, SET_UN,NC}
BAT:= array [1..br_num][1..br_num][1..2] of br_action
BCV:=array [1..br_num] of Boolean

ALGORITHM: BAT_Construction
1.  Alias analysis and identify memory resident values.
2.
3.  Assume each store is a definition of the memory variable, construct
    the reaching definition information.
4.
5.  Foreach load l do
6.    Foreach branch bl whose outcome is inferrable from l's range do
7.      Foreach store s, whose definition reaches l do
8.        foreach branch bs whose outcome can infer s's range and s's
    range subsumes bl's range do
9.          set the action in BAT for bs to bl; mark bl in BCV;
10.       od
11.     od
12.     Foreach load lp, whose use immediately precedes l do
13.       foreach branch blp whose outcome can infer lp's range and lp's
    range subsumes l's range do
14.         set the action in BAT for bs to blp; mark blp in BCV;
15.       od
16.     od
17.   od
18. od
19.
20. Foreach branch br marked in BCV do
21.     Find all branches with definitions (other than the correlated
    loads) to the register used in br. Or other stores(other than the
    correlated ones) to the variable. Mark the action to br as UN in the
    corresponding entries in BAT.
22. od
23. return BAT, BCV
```

Figure 5. Algorithm to construct BAT and BCV.

After a branch is executed, we find out which branches are affected by this branch. For each affected branch, there are four possible actions: SET_T, SET_NT, SET_UN, NC, which means: "set to taken", "set to not-taken", "set to unknown" and "no change" respectively. It is clear that these four represent all possibilities. In addition, as previously mentioned, not all branches need to be checked. If the compiler cannot infer anything in terms of correlation about the branch outcome, the branch can be excluded from checking. Therefore, we set up a vector called Branch Checking Vector (BCV). It only stores which branches should be checked.

The algorithm to construct the BAT and the BCV is shown in Figure 5. Note that the algorithm works on functions rather than on the whole program. It starts with alias analysis to identify all possible memory resident variables that are accessed by each load/store instruction. We first focus on those uniquely aliased variables. The case for multiply-aliased ones will be addressed later. Next, we perform reaching definition analysis for all store instructions after alias analysis. Here, we regard each store as a definition of the variable in the particular memory location.

The main component of the algorithm is a nested loop. The first half of the loop handles correlation between one store and one load. We first need to clarify the relationship between a load/store and a conditional branch. Here we examine each branch whose outcome is inferable from a load's range, which means the branch direction is determinable if the loaded value is in a certain range. For each load, we find the stores that define the same variable and whose definitions reach this load. We then find branches that are related to the store. We want to identify branches whose outcomes infer a value range of the same variable and the range (of the store) subsumes the range of the load. To put it in a simple form, our goal is to discover the follow relationship:

branch bs's direction➔store st's range;
store st's range subsumes load ld's range;
load ld's range➔ branch bl's direction;

In other words, such a relationship indicates that from branch instruction bs's direction we can know bl's direction exactly. Revisit the example shown in Figure 3.c. If the branch in BB1 is taken, it infers y<5. y<5 subsumes y<11. For the load in BB2, we then know that y is less than 11. This infers that the branch in BB4 is taken, since r1=y-1<10. Thus, this formulation is also able to take the following case into consideration: after a variable is loaded into a register, the register participates in further calculations before it is used in a conditional branch.

The second half of the loop handles correlation between two consecutive loads. The two loads must access the same variable, and after the first load the variable must keep alive between the two loads. The reasoning here is similar to the one discovering correlation between one store and one load. The goal is to discover the following relationship:

branch blp'direction➔load lp's range;
load lp's range subsumes load l's range;
load l's range➔ branch bl's direction;

In the example in Figure 3.a, if the path follows BB1→BB2→BB3→BB5, variable y is encountered again. The fact that the branch in BB1 (blp) is taken infers range y<5 (lp's range). y<5 subsumes y<10 (l's range). And y<10 infers that the branch at BB5 (bl) should be taken.

Note that the branch blp could be the same as bl. For example, in Figure 3.a, if the path follows BB2→BB4→BB5→ BB1→BB2, variable x is

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

encountered again. The variable's range is not changed. Therefore the direction of the branch at the end of BB2 should not change.

The last part of the algorithm checks all the definitions other than the correlated loads to the registers used in the branches that are marked in BCV. That is because if there are other paths coming to the branch and the register that is involved in the branch is changed along one of those paths, we must take proper actions accordingly, otherwise the checking mechanism will fail. We mark the branch direction as unknown if such as path is taken since either the compiler fails to derive a range for the redefined value or the range is not related to memory resident variables, which is not interesting in our case.

For multiple-aliased variables, our scheme must be conservative in order to avoid false positives. If the memory access instruction is a load, we simply remove it from further analysis. In this way, no branch will be incorrectly checked, since we do not infer anything from the loaded value. It might be possible that a load comes after this one is now (live range) connected to a load or store before it. It is still fine because the load (use of the variable) will not change the variable. If the memory access instruction is a store, we have to assume all aliases of the stored variable might be defined. Therefore all incoming definitions to those aliases from previous stores must end here. Also, loads that are reached by this store should not be considered to infer anything from this store, because we do not know which memory location the store actually modifies.

5.2 BSV, BCV and BAT Design

In previous sub-sections, we discussed what kind of information should be stored in BSV, BCV and BAT and how to collect the information. Next we need to find an efficient binary representation for those tables. We want to make the tables as small as possible to reduce the space overhead and other associative cost.

The information stored in the tables is with respect to branches identified by PC addresses. Branches are non-uniformly distributed in the code and there could be an arbitrary interval between two adjacent branches. So the natural way to store the information for branches is a hash table. The PC address of a branch is hashed then used as an index into the hash table. However we have to be careful when using hash tables. A common hash table requires tags to handle collisions. If we assume the function contains 1K instructions, the tag will have 10 bits. In other words, the size of the tag could be much larger than the information to be store in BSV and BCV.

To solve the problem, the compiler always finds out a hash function that leads to no collisions for the current function. The compiler achieves this by a trial-and-error method. It utilizes a parameterizable hash function with only shift and XOR operations. The hash function of course is the same one implemented by the runtime system. It first tries different hash function parameters to hash the branches into the optimally sized hash space. If that fails after several tries, the compiler enlarges the hash space and tries again. Obviously the hashing result should depend on the size of the hash space. The hash function has a lower probability to produce collision with a larger space. Since the number of branches in a function is normally not large, in most cases, the compiler can find a proper combination of hash function and hash space quickly. In that way, no tags are required in the hash tables since there is no collision at all. Note that the hash function parameter chosen by the compiler needs to be passed to the runtime system as a part of the information associated with a function.

5.3 Function Calls

The BAT construction algorithm works on functions. However, we have not discussed how to handle function calls. When there are function calls in the function being processed, it becomes a little complicated, since variables might be modified inside the callee function. In other words, the function call site can act like a store to variables, and in some cases we cannot know exactly which variables are modified by the callee function.

We opt to handle function calls in a simple and conservative way. First, we try to prove that the callee function only modifies non-local memory state through its pointer parameters, which should be true for most functions under good programming styles. Local memory state modified by the callee function will be discarded after the function returns thus does not matter. All standard C library function calls are specially handled since we know the exact semantics of those functions. For example, we know strcmp() will not change any non-local memory state and scanf() will only modify dereferenced objects of the second parameter and the following ones. We try to prove the property for user-defined functions by examining the function bodies individually without relying on a full-fledged inter-procedural analysis. If the function modifies non-local memory state through global variables and/or pointer dereferences, to avoid the requirement of an advanced inter-procedural analysis, we can simply act conservatively and assume the call site can modify any variable. After the above analysis, we can convert the function call into a list of pseudo store instructions for the purpose of our correlation analysis. For functions that do not modify non-local memory state, the list is empty. For functions that

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

modify non-local memory state through pointer parameters, we create a pseudo store instruction for each de-referenced actual pointer parameter. For other functions, we create a store that could modify any variable. In that way, we convert the function calls into a list of (possibly multiple aliased) store instructions and how to handle (aliased) store instructions. For library code without source, we currently assume that the callee can modify any variable through pointer parameters. Obviously, the library code itself is not protected.

5.4 IPDS

Upon this point, we have introduced the basic components in our infeasible path detection scheme. Putting them together, the entire framework is called IPDS (Infeasible Path Detection System) and is shown in Figure 6. The infeasible path detection works at the function level. BSVs, BCVs and BATs are constructed on a function basis, and securely protected by the hardware. They are attached to the program binary by the compiler and mapped into a reserved memory space of the program once the program is loaded. The compiler conveys basic information for each function to the runtime system through a function information table. The information includes entry addresses of BSV, BCV and BAT, the entry address of the function, hash function parameters etc.

At runtime, each committed branch is sent to the IPDS. The IPDS first checks whether this branch is marked in the BCV. If it is, the IPDS queues a request to verify if the actual direction of the branch matches the expected direction in the BSV. The IPDS queues a request to update the BSV according to the current branch and the BAT table no matter whether the branch is marked in the BCV.

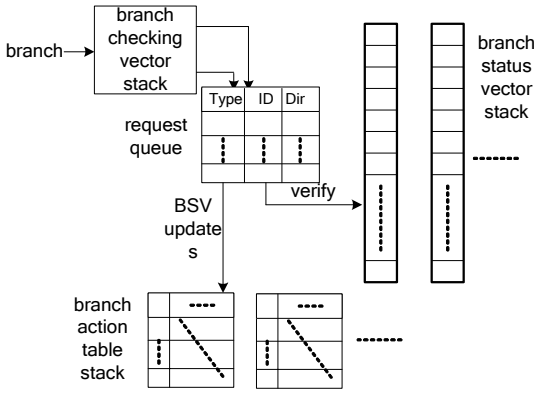


Figure 6. The IPDS framework.

The sizes of BSV, BCV and BAT tables for each function are not fixed. The tables naturally form a stack at runtime. When we enter a new function, its corresponding BSV, BCV and BAT tables are pushed into the top of BSV, BCV and BAT stacks respectively. When the function returns, its BSV, BCV and BAT tables are popped up from the stacks then we can continue the checking based on the caller's BSV, BCV and BAT tables. To save on-chip space, we only keep the top of the stack on-chip but spill the non-active tables to their home location, which is similar to Itanium's register stack engine. The spilled BSV, BCV and BAT tables require proper protection from tampering. The protection can be easily achieved by mapping them to a reserved space. Since only the IPDS is supposed to access those tables and the program is never supposed to access them, any attempted access from the program will be detected and prevented by the processor.

All requests are put in a request queue according to the order in which they are issued. One important observation is that as long as the requests are properly ordered, we can guarantee that the checking is correct. Even if the process of requests gets delayed due to long latency operations such as loading the spilled tables, we can allow the program execution to continue without any delay but queue all the requests in their originally order. Also, it is shown in our experiments that the average checking speed is normally higher than the program execution. In other words, our scheme normally will not slow down the program execution.

The hardware overhead of IPDS includes additional infeasible path detection logic and small on-chip buffers holding BCVs, BSVs and BATs. We believe the hardware overhead is reasonable.

During a context switch, we must save and restore all the tables and vectors. This could be expensive if not done properly especially for the BAT. Actually, we can swap the top of BSV and BAT stacks (around 1K bits) first and let the new process start. Lower layers of stacks are context switched in parallel with the execution of the new process to reduce context switch latency. For further improvements, we can split the BAT into several regions and load the region that is actively used by the other process. To improve locality, hashing should not across regions. Besides, typically only a few important processes should be protected. When context switching to a process that does not require checking, no save/restore is needed.

6. Experiments and Results

All compiler implementations are done in SUIF [25] and MachSUIF [26] research compiler infrastructure. Various phases are enabled including partial redundancy elimination, a graph coloring based regsitera allocator and a publicly available pointer analysis pass for SUIF [27].

The evaluation of the IPDS system consists of two aspects: precision and performance. Good precision

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

means both low false positive rate and low false negative rate. The IPDS achieves a zero false positive rate since it always acts conservatively and only raises an alarm when it is completely sure that an attack is ongoing. Good precision also means low false negative rate, i.e. high detection rate of dynamic infeasible paths due to attacks.

To measure the detection rate, we choose to simulate real attacks that tamper memory state by exploiting program vulnerabilities. We then check whether our scheme is able to detect the simulated attacks. In our experiments, we implement IPDS in an open-source IA-32 system emulator Bochs [24] with Linux installed. We choose 10 server programs with well-known vulnerabilities as benchmarks to perform simulated attack experiments. The server programs and their original well-known vulnerabilities are telnetd(buffer overflow), wu-ftpd(format string), xinetd (buffer overflow), crond(buffer overflow), sysklogd (format string), atftpd (buffer overflow), httpd (buffer overflow), sendmail (buffer overflow), sshd (buffer overflow) and portmap (buffer overflow).

Format string vulnerability allows us tamper an arbitrary memory location, so we can launch as many attacks as we want to the vulnerable program and tamper different memory locations in each attack. But buffer overflow attacks normally tamper a continuous block of memory and only allow us tamper local stack data. Moreover, each of the above server programs only contains a few buffer overflow vulnerabilities. To be able to perform enough number of attacks to measure the detection rate accurately, we manually introduce more buffer overflow vulnerabilities into the server programs originally only having a few buffer overflow vulnerabilities. We further devise our attack to tamper only a (randomly selected) specific local stack location rather than a continuous memory block. Otherwise, we would not be able to make attacks and results independent to each other in the buffer overflow cases. Buffer overflowing a specific location is possible in reality if the attacker knows the original local stack state, for example, by running the same program in his own machine. We make each of our attacks independent to each other to get a better feeling of the detection capability of our system.

Each server program is attacked 100 times independently as explained above. The caused memory tampering may or may not change program control flow. If it does, we check whether the IPDS is able to detect that infeasible path. If it fails, that is a false negative. Our scheme is not designed to handle cases in which memory tampering does not change program control flow.

Figure 7 shows the results of our simulated attack experiments to measure the detection capability quantitatively. We show two results for each benchmark: the percentage of memory tamperings caused by attacks that actually change program control flow and the percentage of memory tamperings caused by attacks that are detected by our scheme. Intuitively, some memory tamperings do not have an effect on the program control flow. Our scheme is not designed to handle those cases. Actually, any scheme focusing on program control flow property will fail to detect those cases. How to detect memory tamperings that do not change program control flow is an open and tough problem [20]. From the results, on average 49.4% of memory tamperings cause changes in the program control flow and the IPDS can detect 29.3% of memory tamperings overall. Thus, the IPDS system is able to detect 59.3% of those memory tamperings that change program control flow. Noticeably, compiler optimizations can remove some correlations, reducing the detection rate.

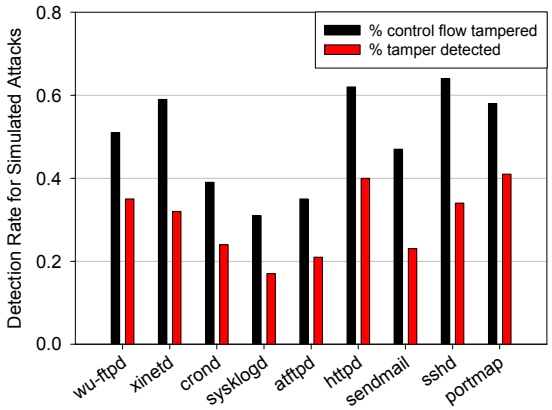


Figure 7. Detection rate for simulated attacks.

Figure 8 shows the average sizes of BSV, BCV and BAT tables in bits. The average size of the BSV table for a function is 34 bits and the average size of the BCV table is 17 bits. Normally they can be packed into a couple of machine words. Now the advantage of our special hash table design should be clear. On the other hand, the size of BAT is much larger due to more complicated data structures. The average size of the BAT for a function is 393 bits.

Next, we focus on the performance aspect of IPDS. The experiments are based on Simplescalar toolset [18]. Each benchmark is simulated in a cycle accurate way by 2 billion instructions. Default parameters for the simulated processor are shown in Table 1. The IPDS hardware is also modeled in Simplescalar.

Notice that, the hardware overhead of IPDS mainly comes from the area taken by BSV, BCV and BAT tables. The on-chip buffers for BSV, BCV and BAT stacks are 2K bits, 1K bits and 32K bits respectively. The on-chip buffers are made large

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

enough such that in most cases they are sufficient to contain the information for branches in the active call chain, even for large server applications. For example, a BCV stack with 1K bits can record information for 1K branches. Assume there is one branch in every eight program instructions and each instruction is 4 bytes. Then the 1K BCV stack can cover 32KB code. The performance cost due to spilling of the stacks is minor. The total on-chip buffer space is only 35K bits, which is very small considering the on-chip cache size has reached several MB in modern superscalar processors. The access latency to the tables is one cycle. Note that we may need to access the BAT table (which implements a link list) several times to handle a BSV update request.

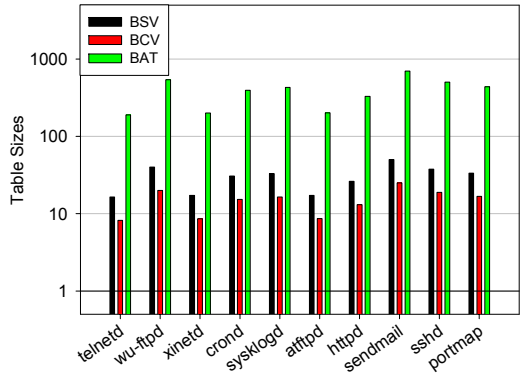


Figure 8. Average sizes (in bits) of BSV, BCV and BAT tables.

In Figure 9, we show the performance for all benchmarks normalized to the baseline case without infeasible path detection. The average performance degradation is only 0.79%. In most cases, the performance degradation is negligible.

We also conducted experiments to measure the time a branch instruction is sent to the IPDS to the time an infeasible path is detected. On average, it is 11.7 cycles. With higher clock frequency, processor pipeline now has over 20 stages. If the checking is initiated at the decode stage, there should be enough time to get response back before the instruction retires. Finally, the compilation time for all benchmarks is up to a few seconds on a Pentium 4 2GHz machine.

7. Related Work

Code injection attack prevention due to buffer overflows, format string attacks etc. has been extensive studied in [2][3][4][5][6]. Code injection attacks are relatively easy to prevent since the attacker must tamper data, inject his code and finally direct the control to his own code. Our work focuses on a new type of attacks in which no code injection needs to be involved thus is more general. On the other hand, the attack tampers the program control flow through memory tampering and manipulates the control flow into invalid/infeasible paths. This new type of attacks will become more and more important when traditional code injection attacks get less important due to the wide deployment of the readily available prevention schemes. We propose an anomaly detection system without false positives that is designed to detect infeasible paths that must be induced by tampered memory data.

Table 1. Default Parameters of the Processor Simulated

Clock frequency	1 GHz	L1 I/D	64K, 2 way, 2 cycle 32B block
Fetch queue	32 entries	Unified L2	512K, 4way, 32B block Latency 10 cycles
Decode width	8	Memory bus	200M, 8 Byte wide
Issue width	8	Memory latency	first chunk: 80 cycles, inter chunk: 5 cycles
Commit width	8	TLB miss	30 cycles
RUU size	128	BSV stack	2K bits
LSQ size	64	BCV stack	1K bits
Branch predictor	2 Level	BAT stack	32K bits

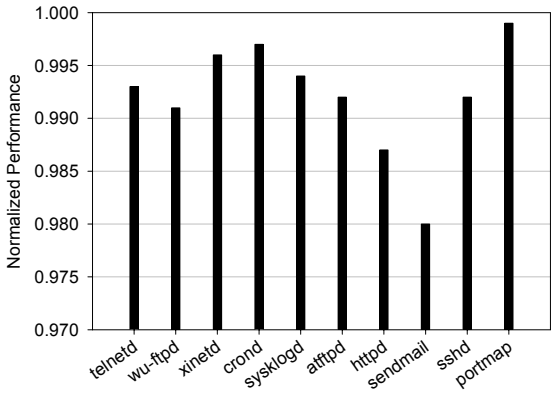


Figure 9. Normalized performance.

Anomaly detection has been extensively studied in the security area [7][8][9][10][11][12][13][28]. As mentioned earlier, they are mostly focused on system call monitoring, the granularity of which is very coarse because the number of system calls is orders of magnitude less than that of branches. However, it is commonly acknowledged that the detection capability of an anomaly detection system largely depends on the monitoring granularity. In contrast with the previous anomaly detection schemes, our scheme monitors program execution at the program control flow level, which is much more fine-grained.

Information flow checking [14][15][19] detects attacks by marking suspicious user input data and all of data depended on the suspicious input data. If some marked data is used in a suspicious way such as computing jump targets, the information flow checking system will raise an alarm. However, if the attack tampers some data expected to decide control flows, the information flow checking schemes will simply fail but our scheme is able to detect a large percentage of

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

those attacks. Therefore, our scheme has better detection capability.

Research has been done [16][17] on how to avoid conditional branches through compiler analysis. The goal is to eliminate branches when the compiler is completely sure about the branch direction. In comparison, our goal is to detect infeasible paths at runtime and our system incorporates both compiler generated directions and runtime information.

8. Conclusion

We propose IPDS to combat memory tampering attacks causing invalid program control flows. In our system, the compiler analyzes correlations between branches and then the collected information is conveyed to the runtime system. The runtime system detects dynamic infeasible program paths by combining the compiler collected information and runtime information. IPDS achieves a zero false positive rate which makes it very practical since false positive is the major obstacle for the deployment of anomaly detection systems. We also show that a high percentage of memory tampering can be detected as long as the tampering actually causes a change in control flow. IPDS requires only a modest amount of hardware resources. The performance penalty due to IPDS is almost negligible. In conclusion, we believe that IPDS would constitute an important component of the overall solution to secure a critical computer system.

REFERENCES

[1] Aleph One, “Smashing the Stack for Fun and Profit,” Phrack volume 7, issue 49.

[2] Nathan Tuck, Brad Calder,George Varghese, “Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow,” MICRO 2004.

[3] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proc. 7th USENIX Security Conference, jan 1998.

[4] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In 10th USENIX Security Symposium, pages 55–66, 2001.

[5] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. SPC-2003.

[6] John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee. A processor architecture defense against buffer overflow attacks. ITRE 2003 2003.

[7] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, "A Sense of Self for Unix Processes," In Proceedings of the 1996 IEEE Symposium on Security and Privacy.

[8] D. Wagner, D. Dean, "Intrusion Detection via Static Analysis," S&P 2001.

[9] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," S&P, 2001.

[10] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, Weibo Gong, "Anomaly Detection Using Call Stack Information," S&P 2003.

[11] Henry H. Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, Barton P. Miller, "Formalizing Sensitivity in Static Analysis for Intrusion Detection," S&P2004.

[12] A.Kosoresow, S.Hofmeyr, "Intrusion Detection via System Call Traces," IEEE Software, vol. 14, pp. 24-42, 1997.

[13] C. Michael, A. Ghosh, "Using Finite Automate to Mine Execution Data for Intrusion Detection: A preliminary Report," Lecture Notes in Computer Science (1907), RAID 2000.

[14] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas, “Secure Program Execution via Dynamic Information Flow,” ASPLOS-04, 2004.

[15] N.Vachharajani, M.J.Bridges, J.Chang, R.Rangan, G.Ottoni, J.A.Blome,G.A.Reis, M.Vachharajani, D.I. August, “RIFLE: An Architectural Framework for User-Centric Information-Flow Security,” MICRO, 2004.

[16] F. Mueller and D. B. Whalley, “Avoiding Unconditional Jumps by Code Replication,” PLDI’92.

[17] R. Bodik, R. Gupta, and M.L. Soffa, ”Interprocedural Conditional Branch Elimination,” PLDI’97.

[18] Doug Burger and Todd M. Austin. “The SimpleScalar Tool Set Version 2.0,” Technical Report 1342, University of Wisconsin--Madison, May 1997.

[19] Jedidiah R. Crandall, Frederic T. Chong, “Minos: Control Data Attack Prevention Orthogonal to Memory Model,” In Proceedings of the 37th International Symposium on Microarchitecture, 2004.

[20] S. Chen, J. Xu, E. C. Sezer, P. Gauriar and R. K. Iyer. "Non-Control-Data Attacks Are Realistic Threats," in Proc. USENIX Security Symposium, Baltimore, MD, August 2005.

[21] Trusted Computing Group. https://www.trustedcomputinggroup.org/home.

[22] Next-Generation Secure Computing Base. http://www.microsoft.com/resources/ngscb/default.mspx

[23] Intel LaGrande Technology. http://www.intel.com/technology/security/.

[24] Bochs: the Open Source IA-32 Emulation Project, http://bochs.sourceforge.net.

[25] Stanford SUIF Compiler Infrastructure, “The SUIF 2 compiler documentation set”, Stanford University, Sep.2000. http://suif.stanford.edu/suif/index.html.

[26] Mach-SUIF Backend Compiler, “The Machine-SUIF 2.1 compiler documentation set”, Harvard University.

[27] R. Wilson and M. Lam, “Efficient context-sensitive pointer analysis for C programs”, PLDI 1995.

[28] Tao Zhang, Xiaotong Zhuang, Wenke Lee, Santosh Pande, "Anomalous Path Detection with Hardware Support," CASES, 2005.

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE