

Memory Protection through Dynamic Access Control

Kun Zhang, Tao Zhang, and Santosh Pande
College of Computing
Georgia Institute of Technology
{kunzhang, zhangtao, santosh}@cc.gatech.edu

Abstract

Current anomaly detection schemes focus on control flow monitoring. Recently, Chen et al. [2] discovered that a large category of attacks tamper program data but do not alter control flows. These attacks are not only realistic, but are also as important as classical attacks tampering control flows. Detecting these attacks is a critical issue but has received little attention so far. In this work, we propose an intrusion detection scheme with both compiler and micro-architecture support detecting data tampering directly. The compiler first identifies program regions in which the data should not be modified as per program semantics. Then the compiler performs an analysis to determine the conditions for modification of variables in different program regions and conveys this information to the hardware and the hardware checks the data accesses based on the information. If the compiler asserts that the data should not be modified but there is an attempt to do so at runtime, an attack is detected. The compiler starts with a basic scheme achieving maximum data protection but such a scheme also suffers from high performance overhead. We then attempt to reduce the performance overhead through different optimization techniques. Our experiments show that our scheme achieves strong memory protection with tight control over the performance degradation. Thus, our major contribution is to provide an efficient scheme to detect data tampering while minimizing the overhead.

1. Introduction

Software suffers from various types of attacks including stack buffer overflow, heap buffer overflow, format string vulnerabilities, and etc. Attackers devise clever techniques to exploit these vulnerabilities. According to the statistics from CERT [1], the number of vulnerabilities reported to CERT increased from 171 to 5990 during the period of 1995 to 2005 showing that the problem is becoming worse instead of going away. Released software could contain various vulnerabilities, many of which are unknown and

therefore systems must be designed to detect attacks based on such unknown vulnerabilities. Intrusion detection system (IDS) is an important solution in this space. There are two types of IDS. The first kind is network based which monitors network packets and matches attack signatures. The other kind is host based that resides in the host and monitors the target system/program directly. We focus on host based IDS.

There are two approaches in host based IDS. The first approach is based on signature checking. It detects attacks by using pre-identified intrusion signatures. Such approaches are accurate but are unable to detect unknown attacks. Also the signatures have to be constantly updated. Another powerful approach is based on anomaly detection which is our focus. It assumes the nature of an intrusion is unknown, but the intrusion somehow deviates from the program's normal behavior.

However, one important observation is that most of the previous anomaly detection work [3][4][5][6][7][8][9][10][11][12] focuses on monitoring program control flows. Thus, one important question to be answered is whether monitoring program control flow is good enough to detect the incidence of an attack. First of all, memory tampering is the primary starting point of attacks. In effect, most previous anomaly detection work tried to detect memory tampering by detecting anomalous program paths caused by the memory tampering. However, during our experiments, we found that a large percentage of memory tampering incidences do not alter program control flows. The big question one may naturally ask is whether we should care about these tampering. Unfortunately, the answer is yes. In [2], Chen et al. showed several real attacks based on data tampering that do not modify program control flows. We will discuss those attacks in detail later, but the conclusion is that it is critical to detect data tampering that does not alter control flow. Such tampering goes undetected under current schemes based on control flow monitoring. Thus, in this work we propose an intrusion detection scheme with both compiler and micro-architecture support that is geared towards

detecting data tampering directly, instead of trying to infer data tampering from anomalous program control flow.

The rest of the paper is organized as follows. Section 2 further motivates our work. Section 3 discusses compiler analyses. Section 4 talks about the architecture support. Section 5 presents our experimental results. Finally, section 6 concludes the paper.

2. Background and motivation

Attacks can be detected by comparing a program's behavior with the one that is expected. Up until now, most anomaly detection research focuses on monitoring the program control flow behavior. Forrest et al. [3] found that system call trace could be considered as quite a distilled execution trace of a program leaving many program structures out. The scheme in [3] tries to detect attacks by checking whether a small segment of the system call trace is normal. Finite State Automata (FSA) based techniques are also proposed to encode the expected system call trace information [8][4][5]. Later work points out that simply checking the FSA for system calls is not sufficient and misses certain attacks. Wagner et al. noticed that impossible paths may result at call/return sites which the attacker can exploit [4]. They proposed a second model called abstract stack model to overcome this limitation. In [6], the authors find that even the abstract stack model can miss important attacks. Thus, in [6] they provide a makeup solution called vtPath to detect certain attacks missed previously by considering more program information. However, [6] also acknowledges that some attacks are still left undetected. [11] proposes incorporating system call arguments into the detection model. Most recently, [9] categorizes system call based anomaly detection systems into "black box", "gray box" and "white box" approaches. [9] further systematically studies the design space of "gray box" approaches and analyzes the importance of monitoring granularity with respect to the accuracy of the system. Their follow up work [10] gives a new "gray box" anomaly detection technique called execution graph, which only accepts system call sequences consistent with the program control flow graph. However due to the limitation of monitoring granularity, execution graph is not able to detect certain attacks.

In summary, monitoring granularity at system call level is not fine enough to detect many attacks in reality. Program shepherding [23] is a scheme dealing with protection mechanisms against control-flow hijacking. It does not rely on system call monitoring but enforces several rules of control transfers. The attacks protected by program shepherding are quite limited. Taint analysis [24] is another scheme that tries to detect illegal overwrites to critical control deciding data by tracking the usage of the

user input data. The schemes only tackle a part of control flow tampering attacks. Recently, Zhang et al. [12] proposed an anomalous path checking technique with the hardware support to monitor segments of dynamic program paths. The idea is similar to the one in [3], but the monitoring granularity achieved is much better and so is the detection strength.

```
FILE * getdatasock( ... ) {
    seteuid(0);
    setsockopt( ... );
    seteuid(pw->pw_uid);
}
```

Figure 1. Attacks without control flow tampering.

One important question is whether monitoring program control flow is good enough. Memory tampering or data tampering is no doubt the primary starting point for attacks. Most if not all attacks start from memory tampering, like buffer overflow attacks, format string attacks. We checked the relationship between memory tampering and control flow modifications. During our experiments, we found that around half of the memory tampering incidences do not alter program control flows. In [2], Chen et al. also showed a large category of real attacks without control flow tampering. One real example is shown in Figure 1.

Wu-ftpd version 2.6 contains a format string vulnerability. Format string vulnerability allows an attacker modify any memory location. Figure 1 shows a piece of code from the Wu-ftpd source code. In normal situations, this piece of code temporarily escalates the privilege using `seteuid(0)` to perform the `setsockopt` operation. Then it will restore the original user privilege after the `setsockopt` operation. But consider a scenario in which `pw->pw_uid` is tampered and is set to 0 through a format string attack. In this case, the second `seteuid` operation will always `seteuid` to 0 again, although it is supposed to restore the original user's privilege. *So the attacker is able to retain root privilege and do all the damage subsequently. Note that there is no control flow tampering involved in this attack.*

Some recent approaches target non-control data attacks. AccMon [25] is a debugging tool based on the observation that a memory location is typically only accessed by a few instructions. AccMon can also be used in combating memory corruption attacks. Their scheme relies on profiling, so it has false alarms. Moreover, since it is based on profiling, an instruction will be regarded legal to access a data object when it does that during any normal program execution, even though under a specific execution, it may be illegal for that instruction to access the data object. This leaves potential holes to be exploited by the attacker. Xu et al. propose a technique to detect memory corruptions attacks in [21]. Their basic assumption is that a randomized program usually crashes upon a memory corruption attack. So they cannot detect attacks that do not result in a process crash. Mondrian memory protection system [22] is a micro-

architectural work that enables fine-grained memory protection. It uses a single, shared address space with access permissions at the granularity of word. At runtime, a permission table is looked up to see if the domain has appropriate access permissions. The access permissions are preset by the user. Mondrian memory protection system provides an important reference in the design of the micro-architecture component of our scheme.

Our work aims to tackle the root cause of memory corruption attacks by detecting corruptions to data directly. We achieve this by providing very fine-grained data access protection. Our solution differs from the Mondrian Memory Protection system; the access permissions are generated by the compiler automatically and access permissions for data objects are managed and changed properly at runtime. Our basic idea is as follows. The compiler first identifies program regions in which certain critical data should remain unmodified obeying the program semantics. Then the compiler sets the data as read-only in those regions. Any attempt to modify the data marked as read-only triggers an alarm. This leads to superior attack detection strength which directly relies on the modification and not on the control flow monitoring.

3. Compiler analyses and optimizations

Before we present the details of our scheme, we first introduce some important definitions.

Definition 1: A *memory object* is a data object defined in a program and resides in the memory. Memory objects include local stack data objects, static global data objects and dynamically allocated heap data objects. Common memory objects are scalar variables, arrays and aggregated structures etc.

Definition 2: An *access permission level* of a memory object is either *writable*, or *read-only*.

Definition 3: A *protection point* is a program point where the access permission levels of some memory objects are changed by setting the corresponding access bits for those memory objects.

Definition 4: A *protection operation* at a protection point is denoted as the action of changing the access permission levels of some memory objects at this point. Generally, there are two types of protection operations – changing from writable to read-only and changing from read-only to writable.

Baseline Scheme

First, we discuss a most basic implementation of our idea. In the baseline scheme, every memory object has a corresponding access bit containing the access permission level for it. There are two potential places to hoist protection points; a protection point could be hoisted just before a store instruction or right after a store instruction.

Assume that a memory object “O” is written by a store instruction. The access permission level for O is changed to writable before the store instruction and is changed back to read-only after the store instruction. Initially, the access permission of all memory objects is set to be read-only. At runtime, whenever a store instruction is to be executed, the hardware checks the access bit table to see whether this instruction violates the memory access permission, i.e. whether there is a write to a memory object whose access permission is set to be read-only. If there is a violation, it alerts the attempted attack. Note that the memory corruption has to be done by some store instructions in a software-based attack. This store instruction could be an existing instruction in the program but writing to a memory object that it is not supposed to. A second possibility is that the store instruction could be injected by an attacker. In either case the tampering is detected by our scheme.

The baseline scheme can prevent memory corruption attacks completely. In other words, *all* memory objects can be protected from being attacked. But it is infeasible to realize this complete protection mechanism due to the very large overhead. Thus, we propose three compiler optimizations to reduce the overhead in a significant way while maintaining the memory protection strength at a fine-granularity level.

Attacks and attack model

Our attack model is based on memory tampering. We assume that all store instructions could possibly be attacking instructions since we try to propose a general solution for detecting all memory tampering attacks. Our technique is not restricted to some specific kinds of attacks, such as buffer overflow attacks and format string attacks. We do not make any assumption about attacks except that they are based on memory tampering. Thus, the experimental results we show later represent the worst case. If our technique is targeted to specific attacks like buffer overflow, format string attacks, the number of possible attacking store instructions will be much smaller and the results will be much better. In other words, the attacking stores will be a subset of all the stores and our scheme protects against any store that is illegally attempting to write to a memory location.

Currently our compiler transformation is only done to the user program and the static library. Memory tampering inside dynamically linked libraries is not protected since dynamically linked libraries are normally not available to the compiler for analysis. Also, our scheme works at the granularity of the memory object level. In other words, the access permission level is always set for the whole memory object. So if an attack tries to modify a different field within an object, the attack will not be detected. The effectiveness of our scheme is determined by the knowledge about store instructions that the compiler can obtain statically. Our scheme is most effective when the

compiler knows exactly which single memory object the store instruction accesses. In those cases, the baseline scheme can provide protection from any memory tampering attack. However, there are cases in which a store instruction could access multiple memory objects at runtime due to pointer dereferences. Since compiler has to be conservative to avoid false positives, in those cases, possible attacks could happen. Please refer to section 3.5 for how we deal with pointer dereferences.

Next we discuss some specific memory tampering attacks and show how our baseline scheme can prevent them.

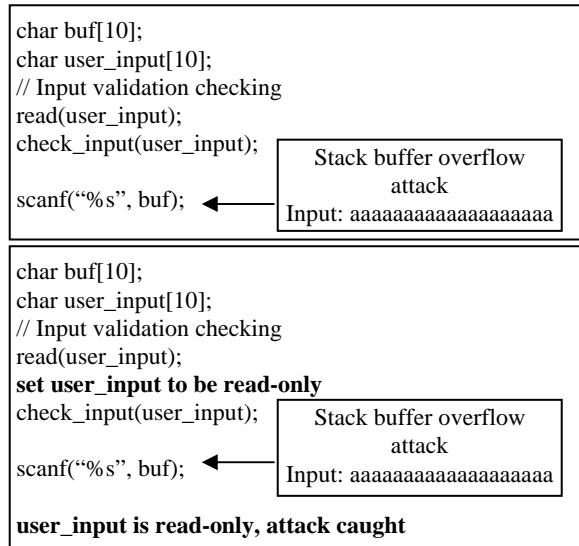


Figure 2. Buffer overflow attack against input string.

Buffer overflows are one of the most common memory tampering techniques. When an attacker injects an overly long input data into a buffer that is not large enough to hold the data, a buffer overflow happens. Our technique could easily detect this kind of attacks. According to program semantics, the store instruction used to store data into the buffer is only legal to access that buffer. It is illegal for the store instruction to access data following the unchecked buffer. This knowledge is obtained by the compiler and our “set to writable” operation will only permit a given store instruction to write into the corresponding buffer. Under the baseline scheme, during the execution of the possible attacking store instruction, the data following the buffer (like a return address etc.) is marked as read-only. When the store instruction tries to over-write the buffer, it tries to modify some data marked as read-only and thus is detected by our scheme. One example is shown in Figure 2

Next, we take the heap corruption attacks against configuration data from Chen’s work [2] as another example shown in Figure 3. Configuration data is a critical piece of data but it is normally only set during the program initialization. The program points where the data gets modified are very limited. Under our scheme, the

configuration data will only be set to writable at those legal program points. The data will be set to read-only otherwise, for example, when heap corruption attacks are launched. Thus, our scheme is able to detect those kinds of attacks.

We tested our technique over all real non-control data attacks shown in Chen’s paper [2], including format string attack against user identity data, heap corruption attack against configuration data, stack buffer overflow attack against user input data, integer overflow attack against decision-making data etc. In all of those attacks, there are some attacking store instructions used to corrupt memory state and they achieve the memory corruption by writing to a memory object that should not be modified by those instructions. In the baseline scheme, those memory corruptions by illegally tampering a memory object are easily detected since a memory object is only marked as writable during the span of the execution of a legal store instruction to the memory object. Normally there are only a few store instructions inside a program that an attacker can attack successfully. So even after our optimizations, as long as the attacking store instruction does not result in a pair of “set to writable” and “set to read-only” operations for the target memory object, the attack will be detected.

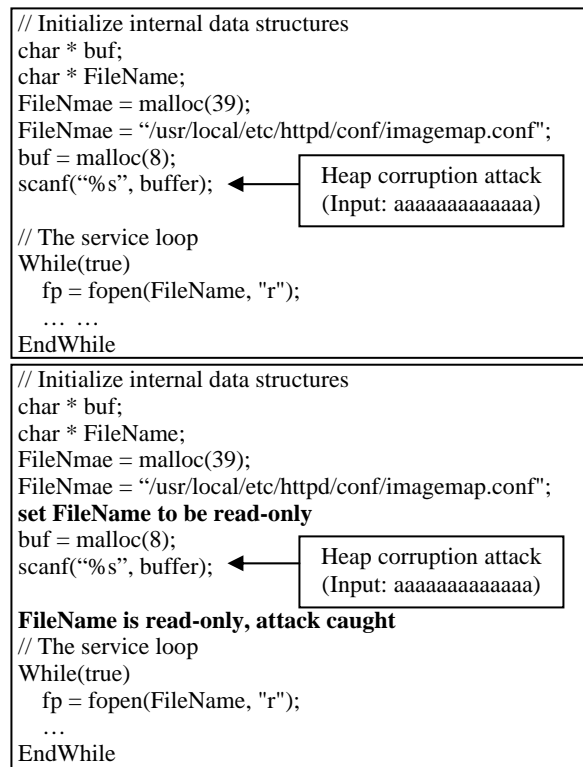


Figure 3. Heap corruption attack against configuration data.

Compiler Framework Overview

The compiler’s task is to analyze the program, optimize the baseline scheme, collect the information about how to set up access permissions, and convey the information to the runtime component. Figure 4

depicts our compiler framework. There are eight steps. First, the pointer analysis is performed trying to statically find out which memory objects a pointer can point to. Second, all store instructions are identified. All program points right before/after store instructions are treated as initial protection points as we stated in the baseline. Third, all write range information is collected to determine where to set the protection (we will talk about write ranges later). Fourth, the hot protection points are hoisted/delayed to cold basic blocks. Then given some performance degradation constraints, the least beneficial protection points are removed. Next, some memory objects are grouped to be protected together. An action table recording protection points and corresponding protection operations is created. So is a pointed-to table used to assist the handling of pointer dereferences. Finally special instructions are inserted into the code to inform the processor when to look up the action table.

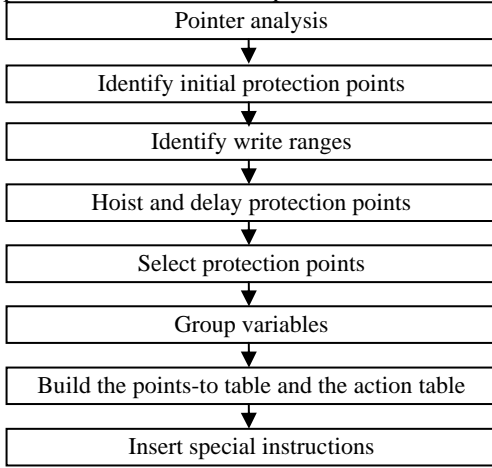


Figure 4. Compiler framework overview.

We found that the code is not very hard to analyze, although in some cases pointer dereferences are involved. We use a context-sensitive and flow-sensitive pointer analysis algorithm [27] to get accurate points-to information and mitigate the impact of pointer aliasing. We found that even after optimizations, our scheme can detect all of these exemplary attacks.

3.1. Write ranges identification

A write range is similar to the live range of a variable, except that the starting point of a write range is a store instruction on a memory object, and the ending point is the next closest store instructions on the same memory object. A write range is used to identify protection pairs. Protection pairs are defined as the corresponding set of “set to read-only” protection operations given a “set to writable” protection operation and vice-versa. The information is

collected to assist the later optimization phases. We utilize the standard framework of webs to identify write ranges. The starting and ending points of a write range are places where protection operations should be performed initially.

3.2. Protection points hoisting and delaying

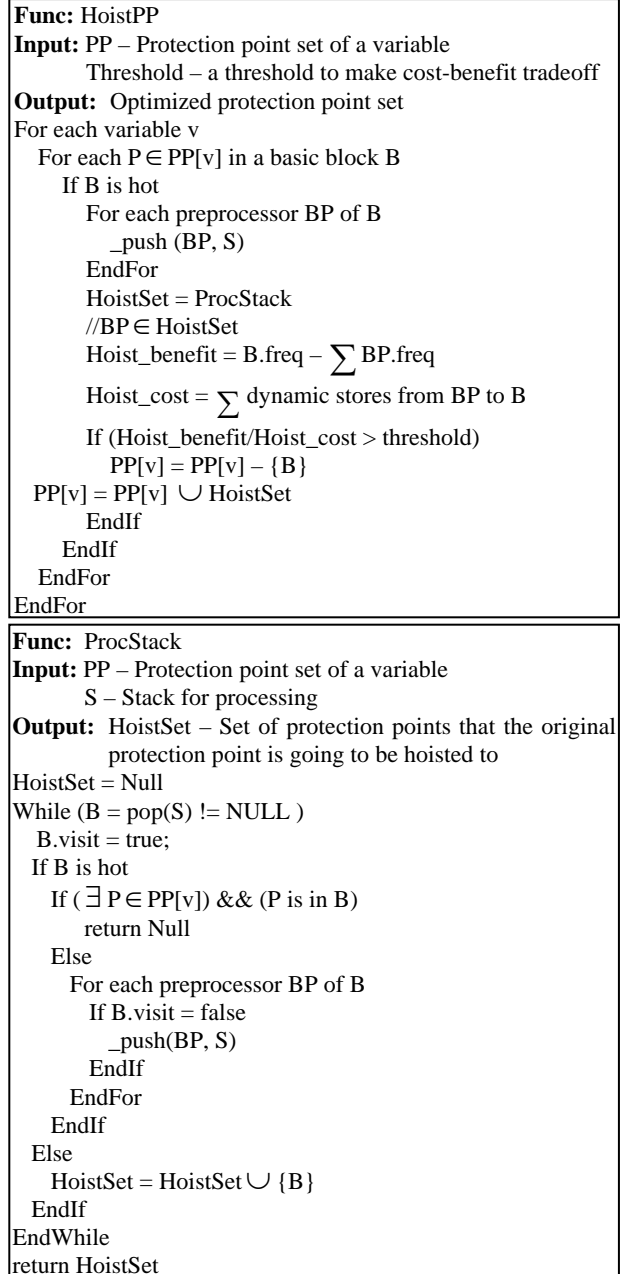


Figure 5. Algorithm for hoisting protection points.

The large overhead of the baseline scheme is mainly due to the fact that some store instructions are executed many times. For example, in some multimedia encoding

applications, the encoding process is done in a major loop executed many times; while other code outside the loop is seldom touched. Hoisting/delaying protection points out of the loop would afford us more efficient mechanisms. This observation motivates us to move protection operations from hot basic blocks to cold basic blocks.

Figure 5 gives the pseudo code for our hoisting algorithm. Starting from a hot basic block containing a protection point, we go back along edges in the control flow graph. Its predecessor basic blocks are pushed onto a stack for further processing. After obtaining the potential locations for hoisting a protection point, we determine whether to actually move the protection point out of the hot basic block based on a cost/benefit analysis. The benefit is denoted as the saving in executing protection operations dynamically. The cost is defined in terms of the security degradation. Hoisting a protection point to a predecessor point means that the object could be corrupted between the two points. So it is not protected in that region. If the ratio of benefit/cost is bigger than a threshold, then the protection point is hoisted out. Currently the threshold is determined by the system manually, which can be optimized using some heuristic solutions.

A similar algorithm is applied to delay protection operations of changing the access permissions from writable to read-only from basic blocks to a cold ones.

3.3. Protection points selection

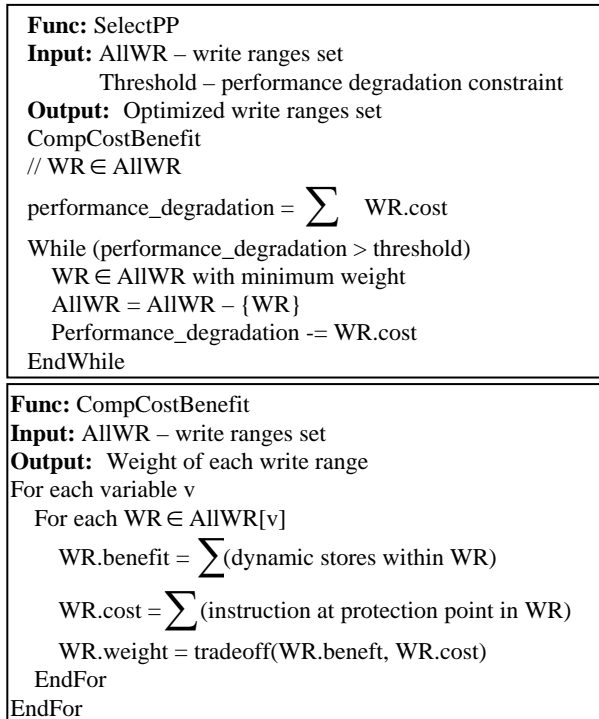


Figure 6. Algorithm for selecting protection points.

Due to a big overhead, completely protecting all memory objects may not be feasible. So how to determine where and which memory object should be protected are the key issues addressed in this section.

We build a cost/benefit analysis model to select protection points. The analysis unit is the set of protection points of a write range. All protection points for a write range are analyzed together since they are related operations. The benefit of removing a write range is denoted as the sum of the dynamically executed store instructions within the write range. The cost of a write range is defined as the sum of the dynamic execution times of all protection points in this write range.

Figure 6 shows the pseudo code to choose protection points. The algorithm analyzes the cost/benefit for all write ranges. The weight of a write range is based on the cost/benefit tradeoff indicating its priority in terms of protection. The weight is equal to WR.benefit/WR.cost. Then, driven by the performance degradation constraint, the algorithm repeats removing write ranges until the degradation requirement is satisfied.

3.4. Grouping protection operations

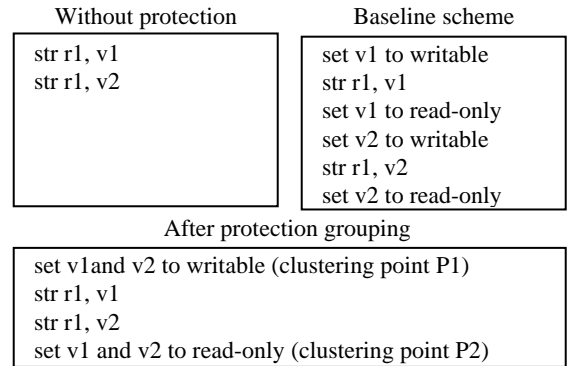


Figure 7. Protection operations grouping example.

In this section, we introduce a compiler technique called protection operations grouping. Normally, each protection point corresponds to one protection operation. However, interesting optimization opportunities exist when multiple protection points performing the same type of operations are clustered together and memory objects they protect are adjacent in the memory. In these cases, instead of incurring one protection operation for each protected memory object, the protection operations can be grouped together as one protection operation for all memory objects at the clustering point. Figure 7 shows a simple example. Assume variables v1 and v2 are adjacent to each other. The start address of v1 (v2) is addr1 (addr2) and its size is size1 (size2). Instead of setting access permission levels of v1 and v2 separately, we can group the operations and set the access permission level of the memory region with start

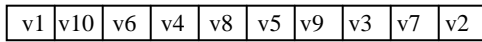
address as addr1 with size as $\text{size1} + \text{size2}$. The grouped protection operation achieves better performance since it reduces the number of dynamic protection operations.

Such optimization opportunities occur when there are multiple protection operations clustered together and when the memory objects protected at a clustering point are adjacent to each other. In our scheme, the compiler tries to move store instructions together as long as their dependencies are still satisfied. Also, after the protection point hoisting/delaying algorithm is performed, many protection operations tend to be grouped at the same program point, like the beginning/end of a cold basic block.

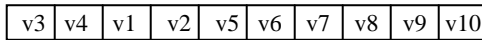
After clustered protection operations are identified, we need to lay out memory objects to exploit optimizations. Figure 8 gives an example showing how the data layout impacts the protection operations grouping. There are nine variables and four clustering points. Figure 8 (b) and (c) give two possible data layouts of these variables in the virtual memory. Data layout 1 scheme only allows performing protections on v5, v8 and v9 together at P4 since only they are adjacent and the protection operations on them can be clustered at the same clustering point (i.e. P4). On the other hand, data layout 2 afford protecting v1, v2, v3 and v4 together at P1; protecting v5, v6 and v7 together at P2; protecting v8, v9 and v10 together at P3; protecting v1, v2, v5 and v6 together at P4; and protecting v8 and v9 together at P4. The two data layouts show substantial differences in grouping protection operation.

set v1, v2, v3 and v4 to read-only (clustering point P1)
 set v5, v6 and v7 to read-only (clustering point P2)
 set v8, v9 and v10 to read-only (clustering point P3)
 set v1, v2, v5, v6, v8, v9 to read-only (clustering point P4)

(a) clustering points



(b) Data layout 1



(c) Data layout 2

Figure 8. An example of data layout.

The pseudo code to determine memory object layout is shown in Figure 9. First, a variable group at a clustering point is identified. All objects with the same type, whose access permissions are changed in the same way at a clustering point form a variable group at the clustering point. Types of an object include initialized global object, un-initialized global object, heap object, and stack object. Access permissions can be changed either from read-only to writable or from writable to read-only. The weight of a variable group is defined as the saving of dynamic protection operations if the protection operations for all variables in the variable group are grouped together at the given point. Then the algorithm starts from the variable group (say CurrVG) with the maximum weight. Note that we only deal with variable groups whose size is bigger than

one since one element group does not provide any information for optimization. Let ProcessedVG be the set of variable groups that have been processed. It is initialized to be null. CommonVG includes variable groups that have common variables with CurrVG. The motivation behind identifying the set of common variables is to maximize linearization of the variables in the group greedily so that more protection operations can be clustered.

```

Func: GroupVar
Input: AllVG – set of all variable groups
Output: Optimized variable groups
ProcessedVG = null
While AllVG != null
  CurrVG = SelectMaxWeightVG
  AllVG = AllVG – {CurrVG}
  If (CurrVG.size > 1)
    CommonVG =
      {VG' | (VG' ∈ AllVG) & (VG' ∩ CurrVG != null)}
    ProcessedVG = ProcessedVG – CommonVG
    If (CommonVG != null)
      ProcessedVG = ProcessedVG ∪ ArrangeData
    Else
      ProcessedVG = ProcessedVG ∪ {CurrVG}
    EndIf
  EndIf
EndWhile

Func: ArrangeData
Input: CommonVG – variable groups having common
  element with CurrVG
Output: Arranged variable group
ListGroup = null
For each VG' in CommonVG
  CommonVar = VG' ∩ CurrVG
  For each v in CommonVar
    CurrVG = CurrVG – {v}
    ListGroup = ListGroup – VG'
    VG' = VG' – {v}
    NewListGroup = CreateListGroup(v, VG')
    ListGroup =
      CombineListGroup(ListGroup, NewListGroup)
  EndFor
EndFor

```

Figure 9. Pseudo code for the grouping algorithm.

Next we show how to use the common variables to clarify the spatial relationship of variables. For each variable group VG' in CommonVG, let CommonVar be the set of common variables of CurrVG and VG'. A variable v in CommonVar is taken out of the variable group. The remaining variables in the variable group form a new variable group. Then v is connected to the new variable group to create a NewListGroup that is in a special form – listgroup. Listgroup is a special structure containing both lists and groups. A group means that variables in it could be grouped together, but it does not inform the exact variable layout, i.e. which variable should be adjacent to which variable in the virtual memory. So the variable can be linearized in many ways. But considering all variable

groups, a variable should be preferred to be the neighbor of another variable to optimize overall protection operations. So a list is designed to represent the linear data layout of the variables. Finally NewListGroup is combined with the original ListGroup to build the current ListGroup.

Besides scalar variables, we can also group array accesses. For example, if an array is written/read in a stride of 2, e.g., it accesses elements 1, 3, 5, 7, 9, 11..., we may group array elements 1, 3, 5 as a group and array elements 7,9,11 as a group. Array access analysis is an important topic that has been extensively studied in [14][15][16][17][18]. With array access information, array accesses can be grouped using standard techniques. For array accesses with a stride of 1, the problem is very similar to loop vectorization. For array accesses with a stride greater than 1, loop scatter-gather can be done first to create a loop accessing the same data with a stride of 1. Any transformation cannot violate the original program dependencies. All the techniques involved are elaborated in [19]. Being able to handle arrays is important to reduce the security cost for the whole program since a large percentage of dynamic memory accesses go to aggregated data structures, most of them being arrays.

3.5. Points-to table

Dealing with pointer dereferences is hard. If the points-to set of a pointer dereferencing store instruction contains multiple possible memory objects, then before the pointer dereferencing store instruction, the access permission levels of all possible pointed-to memory objects have to be set to writable. Also, after the pointer dereferencing store instruction, the access permission levels of all possible pointed-to memory objects have to be set to read-only. This could increase the protection overhead significantly. If we choose not to handle such cases; then we have to give up protecting the dereferencing store instructions and a large number of locations would remain unprotected.

In our scheme, we propose a profile-driven solution to the above problem. Our observation is that due to the limitation of static analysis, although a pointer dereferencing store instruction could possibly access a large set of memory objects according to static compiler analysis, at runtime the number of memory objects actually accessed by a pointer dereferencing store instruction is very limited. The same observation is made in [25]. Thus, we use profiling to identify the most likely accessed memory object by a pointer dereferencing store instruction. Based on this information, a points-to table is created. The points-to table is basically a hash table indexed by the PC addresses of pointer dereferencing store instructions.

When handling pointer dereferencing store instructions, all compiler algorithms proceed assuming that the store instruction will access that most likely accessed memory

object. Thus, the most likely accessed memory object will be properly protected. At runtime, the hardware component checks whether the pointer actually points to the expected memory object by looking up in the points-to table. If the pointer happens to point to a different memory object, the hardware component will skip checking access permissions for this pointer dereference to avoid potential false alarms.

3.6. Action table and special instruction

We now describe how the compiler actually inserts protection operations. The central data structure is an action table. The action table records which action should be performed at a given program point on a given memory object at runtime. The action table is a hash table that uses the PC address of an instruction as its key. We always create a hash table without collisions to avoid the overhead of maintaining a spill list. Each entry of the table has three fields – action, memory object starting address, and memory object size. The action field has only one bit – bit value 1 denotes the action of changing the access permission level from read-only to writable; bit value 0 denotes the action of changing the access permission level from writable to read-only.

A special instruction is inserted at every protection point to inform the processor that at the given program point about some protection operation needs to be done. Whenever the processor encounters such a special instruction, it uses its PC address to index into the action table and to execute the desired action.

Another possibility is to insert instructions to implement the intended protection operations directly instead of recording them in a separate action table. However, the action would require several instructions to implement, which indicates more changes to the standard ISA of the processor and more code space overhead. Using a separate action table plus one special instruction appears to be a better solution.

The action table requires the start addresses and the sizes of memory objects to be known. The important problem is that the information may not be available during compilation time. In general, during compilation, we do not know the address of the local stack objects and the heap objects. For some heap objects, we may not even know their sizes. To solve this problem, the action table must be made writable and the compiler has to insert instructions to fix the action table for stack and heap objects. Such modifications of the action table occur after each dynamic memory allocation and each function entry point. The inserted code will update the action table with the dynamically obtained address and size information.

Making the action table writable results in some security complications. Ideally, the action table should be maintained in a reserved address space and made read-only

to the user program, so that it can only be written by the hardware component of our protection scheme to avoid malicious corruptions from the user program. Thus, how to protect the action table becomes an issue. Our solution is to apply the protection mechanism in a hierarchical way. That is, we use the same method to protect the action table residing in the memory. But in this case, the scale of the problem is much smaller and the problem is much simpler. We know that the action table should only be modified by those fix-up instructions inserted by the compiler. We can regard the action table as one single memory object and its address is predetermined. Thus, the additional action table for protecting the original action table can be easily constructed. It can be put into a reserved address space, thus is not accessible to the user program but only accessible to the hardware component. So it cannot be corrupted by the attacker. Then the original action table can be protected properly.

4. Architectural support

Figure 10 illustrates the architecture support. There are three major data structures. Action table and points-to table are simple small hash tables as explained earlier. They are easy to manage due to their sizes and accesses to them are cached as data in data caches. The access bit table records the access permission level for each memory object. It is allocated in a reserved space and can only be accessed by the hardware component, thus is protected from tampering by the user program. Access bit table is large and is accessed frequently. It has to be carefully managed to avoid significant space and performance overhead. A very similar problem exists in the Mondrian Memory Protection system work [22] and it provides an excellent reference on how to manage this large access permission table. We largely follow their design, regarding each memory object as a memory segment as in their work. We deploy multi-level permission table with mini-SST entries that are elaborated in [22]. To improve performance, a protection lookaside buffer and sidcar registers are also deployed. Utilizing the design in [22] greatly reduces the space and performance overhead of our access bit table.

There are two kinds of operations performed by the hardware component. Upon fetching a protection instruction, the processor uses the PC address of the protection instruction as the key to index the entry in the action table. Then the access bits for the memory object will be set according to the starting address and the size. Another case is when a store instruction is fetched. The processor first checks whether this store instruction is one of the specially handled pointer dereferencing instruction by looking up in the points-to table. If the instruction is in the points-to table, then it is specially handled. If the

memory object dynamically pointed by the pointer does not match with the object recorded in the points-to table, then the processor will skip the access permission checking for this store instruction to avoid any false alarm. Otherwise, the access bit table is looked up to find out whether the memory location is allowed to be written. If the instruction violates the access permission, then an alarm is raised indicating the program is under attack.

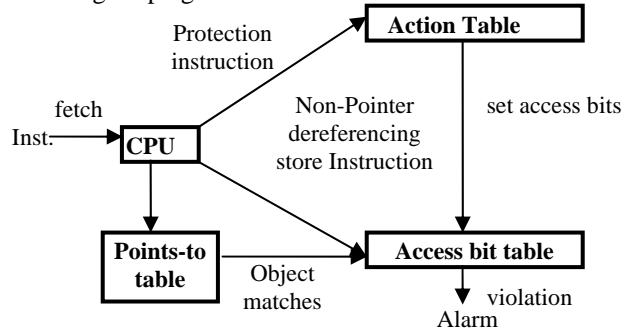


Figure 10. Architecture support overview

False positives

Our technique has zero false positive. We argue it in an informal way. First, the baseline has no false positives. In the baseline, for a store instruction “str R, MO” where R is a register and MO is a memory object, MO is set to be writable immediately before the store instruction. There are two cases. In the first case, the store instruction does not involve any pointer operation. Dynamically when the store instruction is checked, MO should be writable, so there is no false alarm. On the other hand, if the store instruction accesses MO through a pointer, no false positive occurs as we stated in section 3.6 since dynamically we skip checking all store instructions that could result in false positives. For example, for a store instruction str_inst “str R1, (R2)”, where R1 and R2 are two registers, assume its most likely accessed memory object is MO1. Statically, MO1 is set to be writable before str_inst, and an entry of “str_inst→MO1” is created in the points-to table. Dynamically, when str_inst is fetched, it is first checked if the object corresponding to the address in R2 is actually MO1. If so, the system decides to check the access bit table; otherwise, the access bit table is not checked to avoid false positives. Second, our optimization techniques do not introduce false positives. False positives happen when a memory object should be writable but its access permission is set to read-only. In the protection points hoisting/delaying algorithm, a “set to writable” operation is hoisted to the boundary of the “set to read-only” operation on the same memory object. So between the period that a memory object is set to be read-only and the memory object is written, the memory object is set to be writable. Thus there are no false positives. The protection points selection optimization removes a pair of “set to read-only” and “set to writable” protection operations. In other words,

if an operation allowing a memory object to be written is deleted, the corresponding operations that set the memory object to be read-only are also removed. So the memory object is still writable. The protection operations grouping technique obviously does not introduce false positives.

Our compiler compiles standard libraries together with the user program. Standard library functions, such as strcpy, are handled by our scheme. We do not deal with dynamic libraries currently. But we think dynamic libraries can be analyzed separately and conservatively, and relative addressing can be used to encode the compiler collected information. In that way, memory tampering inside dynamic libraries can also be protected to some extent.

5. Experimental results

Our experiments are based on the x86 Pentium architecture. In our experiments, we use 5 different input data sets in training runs to gather the basic block profile which is fed back to compiler passes for memory protections. After the compiler passes are performed, the generated binaries are evaluated by another input data set different from the previous 5 input data sets used in training runs as the test run. The compiler work was implemented using the Machine SUIF compiler [20] that can be used to instrument programs for profiling and carry out various code transformations and optimizations. The benchmarks are chosen from SPEC2000 integer benchmarks and MediaBench embedded benchmarks. The ref input set is used for generating results for SPEC2000 benchmarks. Not all the benchmarks are chosen because Machine SUIF could not compile some of the benchmarks.

5.1. Performance measurement

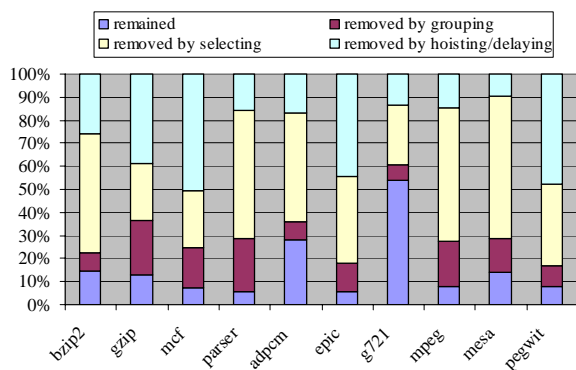


Figure 11. Effects of compiler optimizations.

Figure 11 shows the effects of our compiler optimizations. On average, protection point hoisting/delaying reduces the number of dynamic protection points by 26.1%; protection point selecting reduces the number of dynamic protection points by

39.0%; protection operations grouping reduces the number of protection points by 19.2%. Overall, those optimizations reduce the number of dynamic protection points by 84.3% on average. The reduction results in significant savings in performance overhead.

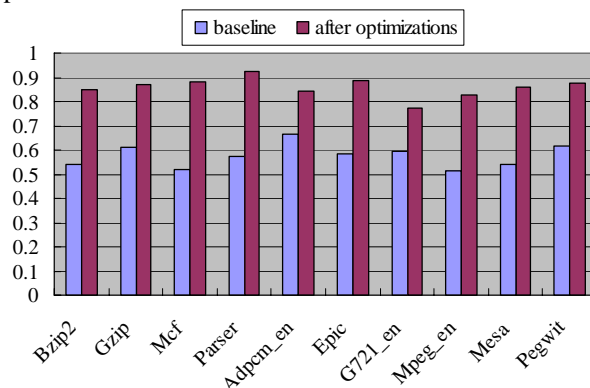


Figure 12. Performance degradation.

Table 1. Parameters of Processor Simulated.

| | | | |
|-----------------|------------|------------------|--|
| Clock frequency | 1 GHz | Branch predictor | 2 Level |
| Fetch queue | 16 entries | BTB | 512 entries, 4 -way |
| Decode width | 4 | PLB | 128 entries |
| Issue width | 4 | L1 I/D | DM, 32K, 1 cycle 32B block |
| Commit width | 4 | Unified L2 | 8way, 32B block 1M (16 cycles) |
| RUU size | 64 | Memory bus | 200M, 8 Byte wide |
| LSQ size | 32 | Memory latency | first chunk: 120 cycles, inter chunk: 10 cycles |

Figure 12 shows the performance degradation. All hardware support modeling is done inside SimpleScalar [13] targeted to x86 [26]. The parameters for the processor are shown in Table 1. Performance numbers are normalized to the original program binary without protection. From the results, the average performance degradation under the baseline protection scheme is 42.5%. The performance degradation mainly comes from the overhead to access the access bit table. Other sources of performance degradation include accessing the action table and the points-to table, and executing the compiler inserted instructions to fix up the action table. The average performance degradation after optimizations is 14.1%. So our optimizations are able to reduce the performance degradation due to data protections significantly because optimizations are designed to reduce the number of dynamic protection points.

5.2. Security measurement

Figure 13 shows the security measurement. The baseline protection scheme can detect all memory corruption attacks like buffer overflows and format string vulnerabilities, since an attack needs a store instruction to do the tampering and that attacking store instruction has to be between the pair of “set to writable” and “set to read-only” operations. This corresponds to the 100% protection shown in Figure 13. Due to the motion and removal of protection points caused by compiler optimizations, some memory locations could be left unprotected. The reason is that another store instruction `str2` may be between the “set to writable” operation and the “set to read-only” operation for `str1` now, and `str2` could be the attacking store instruction and could possibly tamper the memory object to be written by `str1`. We count the number of dynamic store instructions like `str2` and the number of dynamic pointer dereferencing store instructions with an unexpected target memory object, then subtracted those from the total dynamic store instructions to measure the protection strength after optimizations. As per this calculation, a higher number of dynamic store instructions after subtraction means the higher is the protection strength. The results in Figure 13 indicate that the compiler optimizations do not lead to much security degradation. On average, the protection points hoisting/delaying algorithm degrades the protection strength by 3.0%; the protection points selection algorithm degrades the protection strength by 8.1%; the protection points grouping algorithm degrades the protection strength by 0.2%. Overall, after optimizations we achieve an average of 88.7% protection over the baseline scheme, ranging from 82.7% to 98.5%.

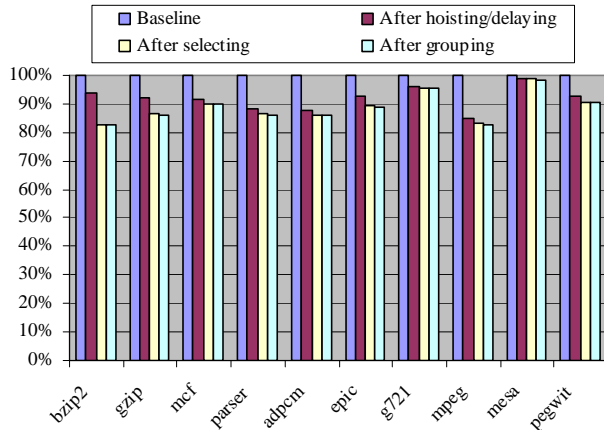


Figure 13. Security measurement.

It should be noted that the above method to measure protection strength represents the worst case. The above method assumes that every store instruction moved into a pair of “set to writable” and “set to read only” operations could be an attacking instruction or an attacking point. In reality, normally a program only has a few possible attacking instructions. For example, for a buffer overflow attack exploiting the `strcpy` function, the store instruction in

the `strcpy` function implementation is the possible attacking instruction. As long as our optimizations do not move that possible attacking instruction into a pair of “set to writable” and “set to read only” operations (which is very unlikely), the security will not be harmed. Thus, our scheme has much stronger detection strength against real-world attacks than represented by the above worst case.

We also tested our scheme against real attacks in [2]. We found that even after optimizations, our scheme can detect all of these attacks. We further performed simulated attacks to our benchmarks. We ran the benchmarks in the simulator and then randomly tamper a memory location by making a store instruction write to a different memory location to see whether our scheme is able to detect the tampering. For each benchmark, we perform such simulated attacks by tampering at 1000 different memory locations. We assume that every store instruction could be an attacking instruction, so again our results represent the worst case since the possible attacking instructions in a program are very limited. Figure 14 shows the percentage of attacks detected. On average 92.7% of the randomly injected memory tampering are detected, which shows that our scheme is very effective in protecting memory tampering attacks.

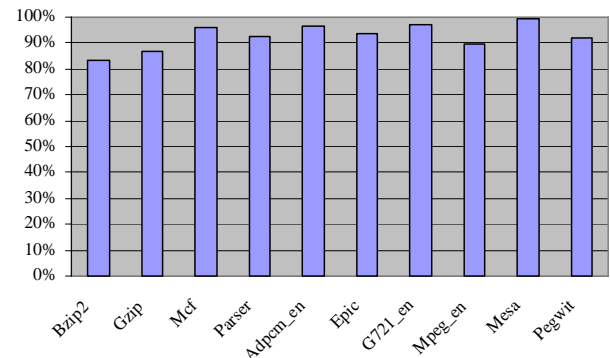


Figure 14. Detected simulated attacks.

5.3. Space cost measurement

The space cost is shown in **Error! Not a valid bookmark self-reference.**. The fields give the sizes of the action table, the access bit table, the points-to table, and the total code size increase of the program due the insertion of protection operations. The additional action table to protect the original action table only requires a little space, so we do not show its size here. We use a multi-level permission table as in [22]. The average sizes of the action table, the access bit table, the points-to table, and the code increase are 96134 bytes, 142 KB, 355 bytes and 51848 bytes respectively. Some access bit table sizes are small, like the `adpcm` encoder and the `g721` encoder, because they have only a few objects and these objects are big. The access permissions for one object are compacted using the Mini-

SST [22] technique, which saves a lot of space. The average size of points-to-set for each pointer dereferencing store instruction is 1.58. The average miss prediction rate of the dynamically accessed memory object for those specially handled pointer dereferencing instructions is 5%. The results show that the space cost of our technique is at most several hundred kilo bytes, which is very acceptable for modern computers.

Table 2. Space cost measurement.

| Benchmark | Action table size (byte) | Access bit table size (KB) | Points-to table size (byte) | Code size increase (byte) |
|-----------|--------------------------|----------------------------|-----------------------------|---------------------------|
| Bzip2 | 14016 | 147 | 472 | 7018 |
| Gzip | 13216 | 159 | 496 | 11520 |
| Mcf | 21248 | 306 | 338 | 29477 |
| Parser | 62272 | 323 | 1222 | 45138 |
| Adpcm_en | 672 | 24 | 30 | 349 |
| Epic | 10784 | 160 | 222 | 5395 |
| G721_en | 3200 | 59 | 100 | 1610 |
| Mpeg_en | 24864 | 89 | 318 | 12436 |
| Mesa | 770976 | 79 | 106 | 385488 |
| Pegwit | 40096 | 77 | 248 | 20057 |

6. Conclusion

In this work, we tackle the problem of detecting memory corruption attacks without relying on program control flow. We propose a compiler and micro-architecture collaboration framework to detect memory tampering. Three optimizations are designed to reduce the performance degradation. By carefully crafting these optimizations, our empirical study shows that the security of the scheme is not reduced much whereas the degradation is reduced significantly. Our experiments prove that our scheme achieves strong memory protection with tight control over the performance degradation.

7. Acknowledgements

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

8. References

- [1] CERT Coordination Center. www.cert.org.
- [2] 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.
- [3] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, "A Sense of Self for Unix Processes," In *S&P'96*, 1996.
- [4] D. Wagner, D. Dean, "Intrusion Detection via Static Analysis," In *S&P'01*, 2001.
- [5] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," In *S&P'01*, 2001.
- [6] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, W. Gong, "Anomaly Detection Using Call Stack Information," *IEEE Symposium on Security and Privacy*, 2003.
- [7] A. Kosoresow, S. Hofmeyr, "Intrusion Detection via System Call Traces," *IEEE Software*, 1997.
- [8] C. Michael, A. Ghosh, "Using Finite Automate to Mine Execution Data for Intrusion Detection: A preliminary Report", *RAID* 2000.
- [9] D. Gao, M. K. Reiter, D. Song, "On Gray-Box Program Tracking for Anomaly Detection", *13th USENIX Security Symposium*, August 2004
- [10] Debin Gao, Michael K. Reiter and Dawn Song, "Gray-Box Extraction of Execution Graphs for Anomaly Detection", *the 11th ACM CCS conf.*, 2004.
- [11] C. Krügel, D. Mutz, F. Valeur, G. Vigna, "On the Detection of Anomalous System Call Arguments", In *Proceedings of ESORICS 2003*, 2003.
- [12] T. Zhang, X. Zhuang, S. Pande and W. Lee, "Anomalous Path Detection with Hardware Support". In *CASES'05*, 2005.
- [13] Doug Burger and Todd M. Austin. "The SimpleScalar Tool Set Version 2.0".
- [14] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". *7th USENIX Security Conf.*
- [15] D. Callahan and K. Kennedy. "Analysis of Interprocedural Side Effects in a Parallel Programming Environment". *Journal of Parallel and Distributed Computing*, 1988.
- [16] B. Creusillet and F. Irigoien. "Exact vs. Approximate Array Region Analyses". In *Lecture Notes in Computer Science*. 1996.
- [17] W. Pugh. "A Practical Algorithm for Exact Array Dependence Analysis". *Communications of the ACM*, 1992.
- [18] R. Triolet, P. Feautrier, and F. Irigoien. "Direct Parallelism of Call Statements". *ACM SIGPLAN Symposium on Compiler Construction*, 1986.
- [19] Y. Paek, J. Hoeflinger, and D. Padua. "Efficient and Precise Array Access Analysis". In *TOPLAS*. 2002.
- [20] Mach-Suif Backend Compiler, The Machine-Suif 2.1 compiler documentation set. Harvard University, Sep. 2001. <http://eecs.harvard.edu/hube/research/machsuiif.html>.
- [21] J. Xu, P. Ning, C. Kil, Y. Zhai and C. Bookholt. "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities". In *CCS'05*, 2005.
- [22] E. Witchel, J. Cates, and K. Asanovic. "Mondrian Memory Protection". In *ASPLOS-X*, 2002.
- [23] V. Kiriansky, D. Bruening, S. Amarasinghe. "Secure Execution Via Program Shepherding". In *USENIX'02*, 2002.
- [24] J. Newsome and D. Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software". In *NDSS '05*, 2005.
- [25] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Yuanyuan Zhou, Sam Midkiff and Josep Torrellas. "AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants". In *Micro'04*, 2004.
- [26] S. Vlaovic, E. S. Davidson, "TAXI: Trace Analysis for X86 Interpretation", In *ICCD'02*, 2002.
- [27] Wilson, R., and Lam, M., "Efficient context-sensitive pointer analysis for C programs", In *PLDI'95*, 1995.