Improving Software Security via Runtime Instruction-Level Taint Checking

Jingfei Kong, Cliff C. Zou, Huiyang Zhou
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816
{jfkong,czou,zhou}@cs.ucf.edu

ABSTRACT

Current taint checking architectures monitor tainted data usage mainly with control transfer instructions. An alarm is raised once the program counter becomes tainted. However, such architectures are not effective against non-control data attacks. In this paper we present a generic instruction-level runtime taint checking architecture for handling non-control data attacks. Under our architecture, instructions are classified as either Taintless-Instructions or Tainted-Instructions prior to program execution. An instruction is called a Tainted-Instruction if it is supposed to deal with tainted data. Otherwise it is called a Taintless-Instruction. A security alert is raised whenever a Taintless-Instruction encounters tainted data at runtime. The proposed architecture is implemented on the SimpleScalar simulator. The preliminary results from experiments on SPEC CPU 2000 benchmarks show that there are a significant amount of Taintless-Instructions. We also demonstrate effective usages of our architecture to detect buffer overflow and format string attacks.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Design, Performance

Keywords

Buffer Overflow, Format String, Hardware Tagging

1. INTRODUCTION

The increasing size and complexity of modern software systems lead to an increasing number of security vulnerabilities. Well-known examples include buffer overflow, heap corruption, format string, integer overflow, etc. By carefully exploiting these vulnerabilities, attackers may cause severe damages to the running process or even ultimately gain the control of victim computers.

During the past decade, numerous schemes have been developed against different kinds of security attacks. Among them, mitigation techniques are one very important defending category since it is always difficult to discover and fix program flaws in advance. As a way of providing additional protection to unsafe systems, mitigation techniques usually try to mitigate the consequence of an attack by stopping the malicious behavior from happening upon attack detection. Although there are a great many of mitigation techniques being developed and deployed, no solution yet can defeat all currently-known exploits[3].

Recently dynamic taint tracking and checking [4], [5] has been widely accepted as a promising mitigation scheme. Based on the observation that attacks are always launched from suspicious I/O channels such as files or network sockets, it seeks to capture the essence of attacks. It treats data from those suspicious input channels as tainted data and keeps track of the tainted data propagation as they may directly or indirectly affect other data values in the program. In order to do so, the processor-memory model is enhanced to keep track of taint information. Based on tainted data tracking, certain taint checking is performed during the runtime execution. It is not until checking fails that the system is to be alarmed. Given the fact that current attacks usually seek to change the control flow of the victim program, one commonly adopted taint checking rule is that control data shall never be tainted [4],[5].

Although control-data attacks are currently dominant, in this paper we focus on non-control data attacks. The reality and applicability of these attacks have already been demonstrated in previous work[1]. It is foreseeable that when control data protection techniques are widely deployed, attackers may have the incentive to bypass those techniques by using non-control data attacks. Previous work from [2] has used taint checking on pointers against those attacks. In other words, if tainted data is used as an address, an alarm is raised. However it has false negative scenarios(See Section 2). Annotating important data structures that should never be tainted could be another approach[2] and it may involve very complex program analysis.

In this paper, we present a generic instruction-level runtime taint checking architecture to prevent non-control data attacks. Under our architecture, instructions are classified

ASID '06 October 21, 2006, San Jose, California, USA.

as either Taintless-Instructions or Tainted-Instructions prior to program execution. An instruction is called a Tainted-Instruction if it is supposed to deal with tainted data. Otherwise it is called a Taintless-Instruction. A security alert is alarmed whenever a Taintless-Instruction encounters tainted data at runtime.

The contribution of the paper is the architectural support for the protection of data integrity through fine-grain instruction-level taint checking. Besides, our taint checking architecture requires minor changes to the original taint tracking architecture and it can complement other taint checking techniques such as pointer taintedness checking[2] to provide a higher degree of security protection. Our preliminary statistical results from experiments on SPEC CPU2000 benchmarks show that there are a significant amount of instructions which are never supposed to deal with tainted data. We also demonstrate the effective usage of our architecture for buffer overflow and format string attack detections on an enhanced SimpleScalar simulator through manual annotation.

The rest of paper is organized as follows: Section 2 presents the background and discusses the related work; Section 3 introduces the design of our enhanced taint tracking and checking architecture; Section 4 discusses our experiments; Section 5 presents the applications of our architecture; and Section 6 concludes the paper.

2. MOTIVATION AND RELATED WORK

2.1 Low-level Vulnerabilities

Low-level software vulnerabilities such as buffer overflow, heap corruption, format string and integer overflow account for the largest portion of CERT advisories[2]. Malicious users may exploit these vulnerabilities to write to arbitrary memory locations with any specified values. For example, an attacker may overwrite values such as function return addresses, jump targets or function pointers. Once control transfer instructions use those values, the control flow may change to the code which the program should not execute. Ultimately an attacker may compromise a host by doing so.

Buffer overflow. It results from writing to a buffer without bounds checking. Once data is stored beyond the boundary of a fixed length buffer, adjacent memory data may be corrupted. It is the most common and exploitable one among all vulnerabilities[7]. There have been enormous efforts put into buffer overflow detection and prevention, including static analysis and dynamic runtime monitoring. A good summary and evaluation of dynamic buffer overflow prevention can be found in [6]. Static analysis may generate too many false warnings or miss errors in the code[17] while dynamic mitigation techniques also have limitations, as highlighted in [3]. Those exploit-focused dynamic mitigation techniques are not completely effective against buffer overflow; those defect-focused dynamic mitigation techniques such as runtime bounds checking are not broadly deployed because of the high performance costs and application compatibility problems.

Heap corruption. Heap data structure for dynamic memory allocation contains user data and heap management data. For performance reasons, user data and management data are normally mixed together in memory. There are certain operations on management data involved with each malloc()/free() call. Heap vulnerabilities, such as heap buffer overflow and double free, allow malicious users to corrupt these management data to cause arbitrary memory write.

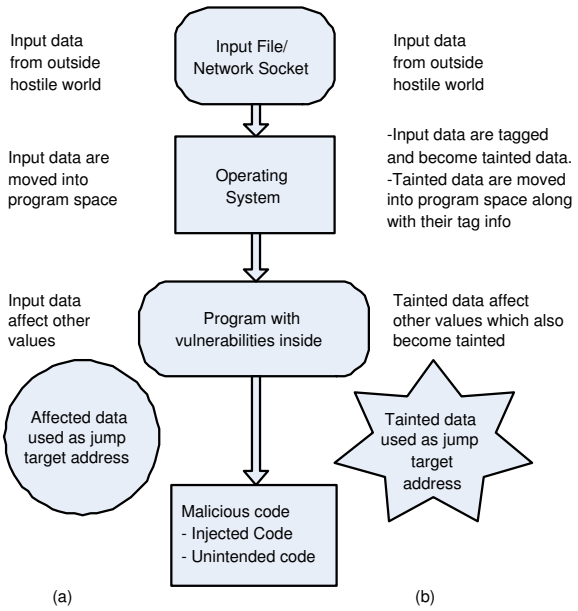

Figure 1: Current Taint Tracking and Checking Scheme

Format string. The use of user input as the format string parameter in certain C functions, such as printf(), may incur security problems. A malicious user may use %s and %x to read sensitive memory content (i.e., information leaking) or even use %n to write arbitrary data to arbitrary memory locations.

In summary, attackers often exploit software flaws to compromise software systems. Low-level vulnerabilities such as those above are all effective examples, which until now people are still struggling to defeat[3]. Although cryptographic methods can be used to protect data confidentiality and integrity, they cannot eliminate software vulnerabilities inside modern systems(For example, the integer overflow as pointed out in [15]).

2.2 Taint Tracking and Checking

Program defects exist and it is always hard to stop attackers from exploiting them. It is also well-known that most current attacks are control data attacks, namely hijacking the control flow of victim programs. It is this final step that attackers follow to break systems and also the final line people can defend their systems from being compromised. For a successful break-in, certain value to be assigned to the intended control data needs to be provided by outside attackers. Based on this observation, *Dynamic Information Flow Tracking*[4] and *Minos* [5] propose using taint tracking and checking technique to stop the final step. Figure 1(a) shows the normal attack procedure and Figure 1(b) shows their basic taint tracking and checking protection mechanism. Here a data value becomes tainted if it is arithmetically derived or simply copied from tainted data. Their scheme is effective because normally the program counter data value is not supposed to be tainted. After their work, there have been several taint-related hardware

```
void do_auth(char* passwd)
{
   char buf[40];
   int auth;

   if (!strcmp("encrypted_passwd",passwd))
     auth = 1;
   else auth = 0;
   scanf("%39s",buf);
   printf(buf); //format string!
A: if (auth)
     access_granted();
}
           (a)

void information_leakage()
{
   unsigned int limit = 50;
   char buf[20];
   int  p[50];
   unsigned int i;

   scanf("%s", buf); //buffer overflow!
B: for (i=0; i<limit; i++)
     printf(" %x ", *(p+i));
}
           (b)
```

Figure 2: Examples of Non-Control Data Attacks

and software proposals adopting the similar taint tracking and checking idea[8], [10], [2], [12] and [11]. In addition, Chen et al [2] proposes to add pointer taintedness checking mechanism, which makes sure that the address value for a memory access is not tainted. Their scheme is based on the observation that non-control data attacks using data pointers are as effective as the well-known control data attacks.

Our proposed architecture is closely related to *TaintCheck* [8] and *Vigilante*[10]. In their work, taint analysis code is added into programs at a fine-grain level by binary instrumentation tools. Through program instrumentation, taint tracking mechanism is implemented and taint checking is performed on the program counter, function call parameters, etc. In comparison, our scheme aims to exploit instruction-level taint behavior and provides a generic fine-grain hardware infrastructure for preventing non-control data attacks. Flexible taint checking policies can be built upon it with minor performance overhead (see Section 6).

2.3 Non-Control Data Attack Examples

As we can see from Figure 1, current taint checking mechanism is only for control data protection. However there are many cases that non-control data are also important for software security. Figure 2 shows two synthetic cases where certain degree of damage can be done by corrupting non-control data. In Figure 2(a), the program is in danger if the critical variable "auth" is overwritten by a format string attack. In Figure 2(b), unexpected extra program data will be revealed out if the counter value "limit" is changed by a buffer overflow attack. Since neither of them involves control data change, attacks in both examples cannot be detected by the original taint checking

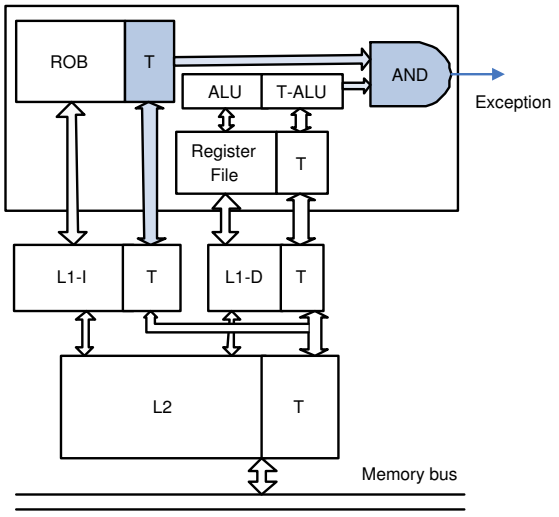

Figure 3: Our Taintless-Instructions-Enhanced Processor-Memory Model

scheme. Pointer taintedness checking[2] may detect the format string attack in example(a) if the attack embeds the target address in the format string itself. However, some format string attacks may use untainted data as the target address[18]. Neither such format string attack nor the buffer overflow in example(b) can be detected by pointer taintedness checking[2] since there are no tainted pointers.

Using our architecture, instructions corresponding to the statements at line A and line B are all Taintless-Instructions since variables "auth", "limit" and "i" are not supposed to be tainted. Attacks in the examples cause "auth" and "limit" become tainted because of taint tracking. Thus they will be detected when the processor executes the instructions corresponding to the line A or the line B. If a format string attack uses untainted data as both the target address and the target value, we resort to taintless instructions in the function vfprintf (See Section 5).

3. DESIGN

Figure 3 shows our Taintless-Instructions-Profile-Enhanced processor-memory model. The data taintedness tracking procedures and associated hardware units are similar to the ones in the original taint tracking scheme [4],[5]. The operating system sets taint tag bits to suspicious input data and forward the data along with tags into program space. The processor fetches, computes and stores data along with taint tags back and forth from the memory. The special T-ALU unit is used for data taintedness tag computation. To support instruction-level taint checking, however, shaded structures have been added. Meanwhile the taint tag (T) for instructions has a different meaning from the taint tag (T) for data, although in the main memory there is still just single extra tag bit for every memory storage. The tag for data indicates whether a value is derived from suspicious outside data. Setting the tag bit to 1 indicates that the value is tainted. The tag for instructions indicates whether an instruction is a Taintless-Instruction. An instruction is called a Tainted-Instruction if it is supposed to deal with tainted data. Otherwise

it is called a Taintless-Instruction. Setting the tag to 1 indicates that the instruction is a Taintless-Instruction. During runtime execution, the taint tags of instructions are buffered inside the processor core, e.g., in the re-order buffer (ROB). Finally, when a committing Taintless-Instruction meets some tainted operand, an exception will be generated through the AND gate.

Whether an instruction is a Taintless-Instruction or Tainted-Instruction must be determined before program runs. It can be collected either through manual annotation, static analysis or dynamic training. A programmer may mark special instructions manually if the data involved is deemed to be very important and also not supposed to be tainted. This is very useful when there is not enough static or dynamic information available. Although its coverage is not as broad as those automatic methods, very important program execution points can be protected. In this paper, we show the effective applications of our architecture mainly through manual annotation. Static analysis can help to identify taint program variables and therefore those corresponding instructions. Dynamic training can help to expand coverage when program source code is not available or cannot provide enough information.

During runtime execution taint checking is carried out in the following four steps:

1. load the collected Taintless-Instructions profile along with program code into the main memory.
2. tag data from suspicious input channels as tainted.
3. track taintedness propagation through execution.
4. raise an alarm when a Taintless-Instruction encounters some tainted operand.

In summary, our proposed scheme defines a new generic instruction-level taint checking architecture for protecting software systems from outside attacks. The instruction-level taint checking can be used to protect against a wide range of attacks, e.g., non-control data attacks as in the examples in Figure 2. It requires minor changes to the original taint tracking architecture.

4. EXPERIMENTS

4.1 Implementation

We implement our scheme on the SimpleScalar processor simulator[9] with the PISA instruction set to study the instruction-level taint behavior. Since our scheme is concerned with protection on both control data and non-control data, it is very important to maintain accurate taint tracking in order to achieve good data coverage. We choose to implement per byte tagging by adding one additional taint bit to each byte. The byte granularity fits into the behaviors of most applications dealing with outside data if not all. It can provide enough accuracy without sacrificing much performance. During program load, taintedness bits corresponding to program code are initialized according to the collected profile and taintedness bits for program data are cleared. Those data coming from certain I/O system calls like READ, RECV, etc. are tagged as tainted. As for taintedness propagation, the common logic is that the destination taintedness bit is the bitwise OR of the corresponding taintedness bits from source operands. In addition, special attention has been paid to certain ALU operations. Shift instructions cause taintedness bit to shift correspondingly. Any untainted byte of value 0 with AND instructions causes taint bit 0 for the outcome byte. Any untainted byte of value "0xFF" with OR instructions causes taint bit 0 for the outcome byte. These rules are borrowed from [2]. Finally a security exception is raised whenever a Taintless-Instruction meets some tainted operand.

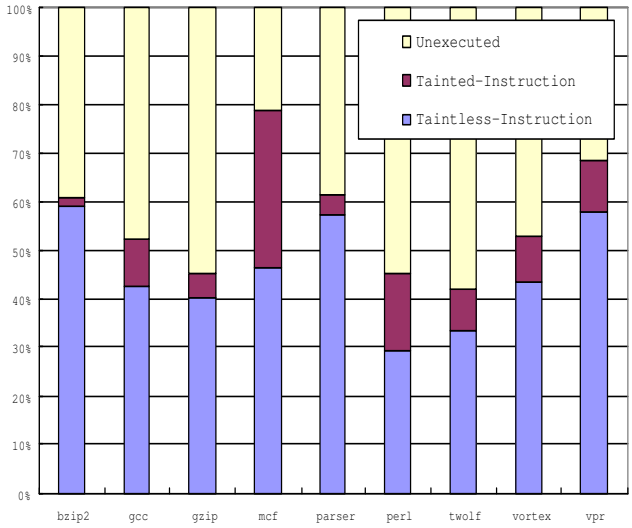

Figure 4: The Percentages of Taintless-Instructions and Tainted-Instructions in SPEC CPU2000 CINT

4.2 Preliminary Results from SPEC CPU2000

Intuitively, in most programs, there is a large portion of program code dealing with untainted data. Our architecture will be of limited use if the majority of program instructions are Tainted-Instructions. To validate our observation, we have used the SPEC CPU2000 CINT benchmarks for experiments and run them to completion using the reference inputs. The instructions are classified as either Taintless-Instructions or Tainted-Instructions. If there are multiple reference inputs, we combine the multiple classifications together. If an instruction is identified as a Tainted-Instruction in one run while a Taintless-Instruction in another, the combined result is a Tainted-Instruction. Figure 4 shows the final statistical results. Because of limited number of reference inputs, many instructions of the programs have not been exercised. The statistical results may not be complete. However it shows that there are a significant amount of Taintless-Instructions (average approximately 46% Taintless-Instructions against 11% Tainted-Instructions).

5. APPLICATIONS

By augmenting program code with extra taint information, our design provides an instruction-level taint checking architecture. It can be used to detect various security attacks at different levels. In this section, we demonstrate the usage of our architecture on detecting buffer overflow

```
void  vuln_stack_function_ptr(int choice){
         volatile unsigned int pad_begin = 0; //padding data
         long stack_buffer[BUFSIZE];
         volatile unsigned int pad_end = 0;   //padding data
         void (*stack_function_pointer)(void);
          . . .
00401cb8 memcpy(stack_buffer, overflow_buffer, overflow+4);
00401cc0 pad_begin++;          //access padding data
   |     pad_end++;
00401cf8
         /* Function call using the function pointer */
00401d00  (void)(*stack_function_pointer)();
00401d08
```

(a)

```
00401cb8 <vuln_stack_function_ptr+2a8> jal 00404e80 <memcpy>
00401cc0 <vuln_stack_function_ptr+2b0> lw $v0[2],32($sp[29])
00401cc8 <vuln_stack_function_ptr+2b8> addiu $v0[2],$v0[2],1
00401cd0 <vuln_stack_function_ptr+2c0> sw $v0[2],32($sp[29])
00401cd8 <vuln_stack_function_ptr+2c8> lw $v0[2],32($sp[29])
00401ce0 <vuln_stack_function_ptr+2d0> lw $v0[2],104($sp[29])
00401ce8 <vuln_stack_function_ptr+2d8> addiu $v0[2],$v0[2],1
00401cf0 <vuln_stack_function_ptr+2e0> sw $v0[2],104($sp[29])
00401cf8 <vuln_stack_function_ptr+2e8> lw $v0[2],104($sp[29])
00401d00 <vuln_stack_function_ptr+2f0> lw $v0[2],108($sp[29])
00401d08 <vuln_stack_function_ptr+2f8> jalr $ra[31],$v0[2]
```

(b)

Figure 5: Buffer Overflow Attack Detection

and format string attacks through manual annotation.

5.1 Buffer Overflow

Figure 5 shows the way to use our architecture to detect buffer overflow. One padding data "pad_begin" is added right before the buffer and one padding data "pad_end" is added right after the buffer. Then the simple operation of "++" is inserted to access those padding data after the suspicious buffer operation ("memcpy" at 0x00401cb8). The instructions for "++" are from 0x00401cc0 till 0x00401cf8 and they are all Taintless-Instructions. Once there is a buffer overflow in "memcpy", tainted input data will be copied across through "pad_begin", "pad_end" and cause them to be tainted. Therefore an exception is raised to signal the danger. Note that the statements "pad_begin++" and "pad_end++" are used to illustrate the padding data accesses at the source code level. The compiler-generated padding data accessing instruction may simply be a single load instruction rather than the instruction sequence presented in Figure 5b.

The example code shown in Figure 5a is manually modified from the testbed of twenty buffer overflow attacks developed by John Wilander[6]. Besides padding data and corresponding access code, we construct the "overflow_buffer" through I/O instead of the original array copy so that "overflow_buffer" data are tainted. "jmp_buf" related code is also changed since the SimpleScalar package provides the old version of "jmp_buf" data structure. Using our Taintless-Instructions-enhanced SimpleScalar simulator, we have tried all attack forms in the testbed using similar modifications and successfully identified all the attacks.

Besides protecting buffers in the stack and data segment, heap buffer overflow can be effectively detected in a similar way by modifying the structure of memory chunks and the management routines. The in-band heap management data is enclosed by two padding data. Any double linked

```
int
DEFUN(vfprintf, (s, format, args),
  register FILE *s AND CONST char *format AND va_list args)
{
        /* Pointer into the format string.  */
        register CONST char *f;
          . . .
00400cb0  f = format;
00400cb8
00402f68  while (*f != '\0')
00402f70
00402f78 {
00400cc0   . . .
```

(a)

```
00400cb0 <vfprintf+1e0> addu $t0[8],$zero[0],$s0[16]
00400cb8 <vfprintf+1e8> j 00402f68 <vfprintf+2498>
00400cc0    . . .
          . . .
00402f68 <vfprintf+2498> lb $v0[2],0($t0[8])
00402f70 <vfprintf+24a0> lbu $v1[3],0($t0[8])
00402f78 <vfprintf+24a8> bne $v0[2],$zero[0],
                           00400cc0 <vfprintf+1f0>
```

(b)

Figure 6: Format String Attack Detection

list manipulations which take place during the heap management process are prefaced with Taintless-Instructions accessing those padding data. This idea is similar to the ones in [16]. To use our architecture to effectively detect all buffer overflow though, it is important to access the correct padding data right after suspicious buffer operations.

Compared to other canary protection schemes such as Stack Guard[13], ProPolice[14], the above scheme based on our architecture has both performance and security advantages. It is known that Stack Guard and ProPolice are based on the guard canary and the checking code. With the compiler help, our scheme requires a much fewer number of checking instructions to be executed. Instead of the need for loading the canary, computing values for complex canary schemes, comparing the values, our scheme will only require one load instruction. Our scheme is more secure in the fact that the padding data are not required to be secret. In comparison, the canary must remain secret for other canary schemes to be effective. Otherwise attackers can overwrite the canary with a carefully-chosen value without being detected. For example, if the original canary is a random number, attackers can perform buffer overflow successfully by overwriting the canary with the same random number. Attacks such as above are possible given the existence of vulnerabilities which may lead to information leakage. Under our architecture, the padding data would be checked by Taintless-Instructions. Even when it is overwritten with a carefully-chosen value by tainted data, the attack will still be detected.

In summary, our architecture can provide better support for canary schemes against buffer overflow attacks. In addition, if combined with control data taint checking and pointer taintedness checking, we expect that it is very difficult to bypass the final checking code without being caught. Thus even some buffer overflow attack just corrupts non-control and non-data-pointer values, it can still be detected.

5.2 Format String

Format string vulnerability may occur when user inputs

are used as format arguments in a printing function. The examples include sprintf, snprintf, fprintf, vprintf, vsprintf, vsnprintf, vfprintf, syslog, and vsyslog, etc. Among them, vfprintf is the basic function on which the other wrapper functions are based. Figure 6(a) shows a part of source code in vfprintf from the C library in the SimpleScalar simulator package. Pointer "f" points to the format string and is used to interpret all kinds of supported specifiers including the famous "%n" in the "while" loop. To detect the attack, the instruction at "00402f68" is marked as a Taintless-Instruction. In other words, user inputs are not allowed to be used as format strings. Each time a tainted input is used as a format string, it will raise an alarm. Similar actions can also be taken to single out the "%n" write attacks. In summary, our architecture enables taint checking on arguments of critical functions such as vfprintf, system calls to provide stronger security protection.

6. DISCUSSION AND FUTURE WORK

6.1 Overhead to Taint Tracking Architecture

Since our scheme is based on the original taint tracking architecture, there is no change to the memory system and the processor pipeline dealing with program data and corresponding taint tags. The overhead is related to the taintedness bits associated with instructions, which are the shaded structures and related data pathes in Figure 3. All additional computation and propagation happen in parallel, not on critical path and should not affect the number of pipeline stages and the cycle time. The additional software processing will only occur during the load of program code. Thus our scheme would introduce minor overhead to the original taint tracking architecture, while it can be used to provide a higher degree of security protection.

6.2 Collection of Taintless-Instructions Profile

The Taintless-Instructions profile is essential to provide security protections based upon our architecture. We have already shown in this paper that a manually-annotated profile can be used to protect some security critical data structures. However, the security coverage provided by this method is limited. The preliminary results from experiments on SPEC CPU2000 benchmarks in Section 4 have shown that there are a significant amount of Taintless-Instructions. Once those Taintless-Instructions are identified, higher security coverage can be achieved. We are currently working on analyzing program characteristics of real applications and using automatic methods such as static analysis, dynamic training to identify Taintless-Instructions.

6.3 Vulnerability and Shortcoming

Under our architecture, each instruction is either a Taintless-Instruction or a Tainted-Instruction. However a Tainted-Instruction does not always deal with tainted data due to the dynamic feature of program executions. Thus our scheme may fail to provide protection to certain untainted data which is handled by Tainted-Instructions. Also the Taintless-Instructions profile does not provide information about the pointer taintedness. Even combined with pointer taintedness checking scheme[2], our architecture still faces some challenges as highlighted in [18]: code overwrite attacks for writable memory region and vulnerabilities coming out of translation tables. Although the first challenge can be solved through instruction taintedness checking, our current Taintless-Instructions profile is not applicable in that case. Further investigation is under way to evaluate the effectiveness of the combination.

6.4 Conclusion

This paper proposes a new generic instruction-level runtime taint checking architecture. It requires minor changes to the original taint tracking architecture and it can complement other taint checking techniques to provide a higher degree of security protection. The effective usages of our architecture for buffer overflow and format string attack detections are presented as examples in the paper.

7. REFERENCES

[1] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravi Iyer. "Non-Control-Data Attacks Are Realistic Threats". In Proceedings of USENIX Security Symposium, August 2005.

[2] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, Ravi Iyer. "Defeating Memory Corruption Attacks via Pointer Taintedness Detection". In Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN), June, 2005.

[3] Jonathan Pincus and Brandon Baker. "Mitigations for Low-level Coding Vulnerabilities: Incomparability and Limitations". http://research.microsoft.com/users/jpincus/mitigations.pdf, 2004.

[4] G. Edward Suh, Jae W. Lee, David X. Zhang, and Srinivas Devadas, "Secure Program Execution via Dynamic Information Flow Tracking". Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI), Oct 2004.

[5] Jedidiah R. Crandall, Frederic T. Chong. "Minos: Control Data Attack Prevention Orthogonal to Memory Model". in the 37th International Symposium on Microarchitecture, December 2004.

[6] John Wilander and Mariam Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention", In Proceedings of the 10th Network and Distributed System Security Symposium (NDSS'03), February 2003.

[7] Jonathan Pincus and Brandon Baker. "Beyond stack smashing: Recent advances in exploiting buffer overruns". IEEE Security and Privacy, 2004.

[8] James Newsome and Dawn Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software". In Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05), February 2005.

[9] D. Burger and T. M. Austin. "The Simplescalar Tool Set Version 2.0". Technical Report, Computer Science Department, University of Wisconsin-Madison, 1997.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms", 20th ACM Symposium on Operating Systems Principles (SOSP), Brighton, UK, October 2005.

[11] Georgios Portokalidis, Asia Slowinska and Herbert Bos. "Argos: an Emulator for Fingerprinting Zero-Day

Attacks". In proceedings of ACM SIGOPS EUROSYS 2006, Leuven, Belgium, April 2006.

[12] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield and Steven Hand. "Practical Taint-based Protection using Demand Emulation". In proceedings of ACM SIGOPS EUROSYS 2006, Leuven, Belgium, April 2006.

[13] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, A. Grier, S. Beattie, P. Wagle, and Q. Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In Proceedings in the 7th USENIX Security Symposium, January 1998.

[14] Etoh, Hiroaki and Yoda, K. "Protecting from stack-smashing attacks". http://www.research.ibm.com/trl/projects/security/ssp/main.html (2004).

[15] SSH CRC-32 Compensation Attack Detector Vulnerability. http://www.securityfocus.com/bid/2347

[16] W. Robertson, C. Kruegel, D. Mutz, F. Valeur. "Run-time Detection of Heap-based Overflows". In the Proceedings of the 17th USENIX Large Installation Systems Administration Conference (LISA). October 2003, San Diego, CA USA.

[17] Olatunji Ruwase and Monica S. Lam. "A Practical Dynamic Buffer Overflow Detector". In Proceedings of the 11th Annual Network and Distributed System Security Symposium, February 2004.

[18] Michael Dalton, Hari Kannan, Christos Kozyrakis, "Deconstructing Hardware Architectures for Security". 5th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) at ISCA, Boston, MA, June 2006.