DOG: Efficient Information Flow Tracing and Program Monitoring with Dynamic Binary Rewriting

Qin Zhao Winnie W. Cheng Bei Yu Scott Hiroshige
{zhaoqin, wwcheng, beiyu, skhirosh}@mit.edu

Abstract

In this paper, we present DOG (Dynamic Observant Guard), an efficient information flow tracing and program monitoring system. Our system is based on dynamic binary rewriting. We proposed a number of novel techniques such as memory mapping to reduce overhead in tracing of data and monitoring of program execution. We have demonstrated that our system is effective in protecting against various kinds of exploits. We also evaluated our system performance with SPEC2000 INT [5] benchmark. Our system achieved an average slowdown of 5.5 times compared against the native execution, offering significant improvements over similar tools.

1 Introduction

Critical vulnerabilities and security exploits are the norm in todays computer systems. The reality is that software is buggy and computing devices must develop an immunity against these attacks. In a summary of vulnerabilities in the Red Hat operating system, buffer overflow and the class of overwrite attacks were identified as the number one source. There were also a number of temporary file creation and information leak vulnerabilities. A security tool should protect against the common critical attacks. In addition, it should provide a protection mechanism for sensitive data even when the system is compromised.

A technique that has been suggested to counter exploits from most critical vulnerabilities is called information flow tracing. Dynamic information flow tracing is a technique to track the propagation of tainted data during program execution. The taint data can be user input, input from a network socket, or data read from files. Information flow tracing is useful for restricting the use of untrusted data as well as identifying sensitive data that can be handled more meticulously.

Currently, there are three ways to track taint data: (1) interpretor-based approach such as Perl's taint mode [3], (2) simulator-based approach [17, 10], and (3) instrumentation-based approach [14]. All these tools suffer from significant performance overhead prohibiting their use for realtime applications. For example, TaintCheck [14] demonstrated a slowdown of over 30 times compared with native execution. The overhead is mainly due to a large number of extra operations being performed to maintain and propagate the taint status for each application operation.

Based on DynamoRIO [8], our system, DOG, uses an instrumentation approach to dynamically trace the propagation of taint data. DOG consists of a number of optimizations to keep the overhead low. It is able to protect against a broad set of exploits such as format string and buffer overflow. In addition, DOG provides a generic cryptographic interface to protect the sensitive data stored in temporary files.

Our approach has the following attractive properties:

- **Language independent.** Since our tool operates on the binary level, it can be used by applications written in any programming language. This is especially useful in protecting legacy software from being attacked without modifying the source code or recompilation.

- **Comprehensive tracing.** Our tool can trace the taint data during the execution of the application code as well as all the shared libraries.
- **Real-time usage.** Our tool is the first of its class that achieves good enough performance for practical use.

The remainder of the paper is organized as follows. Section 2 gives an overview of our system. Section 3 describes the details of our design decisions and implementation techniques. Experimental results are discussed next and in Section 4. Section 5 gives an overview of related work and previous attempts at the problem. The paper concludes our work in Section 6 and addresses potential issues to be further explored as future work.

2 System Overview

Our information flow tracing and monitoring system takes on an object-oriented model that provides a flexible and extensible interface for adding safe-guard operations and for detecting suspicious behavior. It consists of a front-end that provides a user-friendly interface to specify configurations for individual applications and a back-end that invokes the application within a controlled execution environment. Figure 1 shows an overview of DOG.

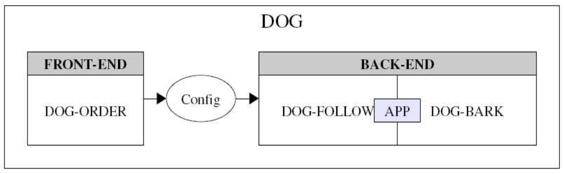


Figure 1: DOG System Overview

The system administrator specifies TaintObjects through the DOG graphical user interface (DOG-ORDER). These objects may be coarse-grained such as all input from network interface or fine-grained such as a specific global variable in the application. Each object is associated with a set of propagation policies and actions. Some examples of actions include types of exploit detection with varying degrees of alert level and encryption/decryption capabilities.

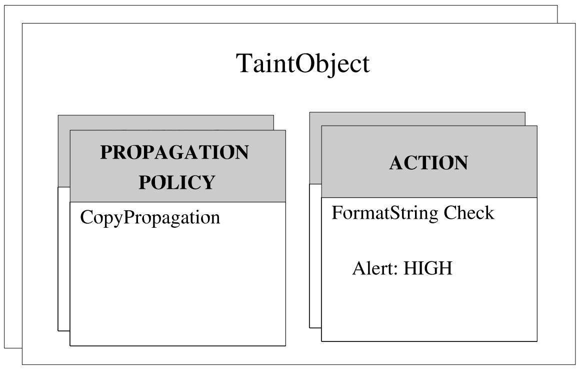


Figure 2: TaintObject

The application is invoked within our back-end framework, a DynamoRIO-based client that implements our efficient information flow tracing algorithms (DOG-FOLLOW) and executes the appropriate protection mechanisms (DOG-BARK). Based on the object and properties specified in the configuration, DOG-FOLLOW enables propagation policies such as copy and arithmetic propagation on the defined object. DOG-BARK performs security checks for possible exploits resembling buffer overflow, format string and other attacks. It also guards against the overwrite of taint objects to files and provides encryption/decryption on file streams. The next section describes the details of the implementation techniques for DOG back-end.

3 Design and Implementation

3.1 Taint Tracing

As possible starting points of taint tracing, the administrator can specify the data from specific input device or network socket, or identify specific memory locations as tainted. Our taint tracing tool tracks the propagation of tainted data at runtime. There are different behaviors of taint propagation, which are summarized in subsection 3.1.1. In subsections 3.1.2 and 3.1.3

we describe the potential overhead for tracing tainted data and how we manage to minimize the overhead in our design and implementation. In subsection 3.1.4 we describe how we check the taint data in sensitive locations to prevent or detect attacks. Finally in subsection 3.1.5 we give some discussions about our design.

3.1.1 Taint Tracing Policy

The *residence* of taint data is a memory location or a register. Taint data can be propagated among different residences in various ways, which fall into following four categories.

1. *Copy Propagation*: Taint data is copied from one residence to another residence.
2. *Arithmetic Propagation*: Taint data is transformed from input operand to the output of a mathematic or logical computation.
3. *Address Propagation*: Taint data can be used to calculate a memory address, which could propagate the sensitive data through a table lookup approach.
4. *Control Propagation*: Taint data may also be propagated through deliberate control transfer. For instance, code like `if(x == 1) y = 1; else if(x == 2) y = 2; ...` may propagate the taint data `x` to `y`.

Our default tracing policy is: the output (destination) data is tagged as tainted if and only if any of the input (source) data is tainted, which covers propagation type 1 and 2. We did not trace propagation type 3 because it is prone to cause false positives. However, we think this type could be treated as an option for users to specify if they need higher security level for their programs. For propagation type 4, naive taint propagation of control-dependent data can lead to a large percentage of false positives. This is a fundamental problem faced by previous approaches [14, 9]. Instead, our tool allows users to specify the critical places in their program where control flow should not be influenced by tainted data.

3.1.2 Design

Basic Idea

Similar to Memcheck [15], we associate each byte of memory and general purpose register with a shadow memory byte to indicate its taint status: 1 represents tainted and 0 represents clear. When an application instruction is executed, a series of operations is performed to propagate the taint status from the input (source) data shadow memory to the output (destination) data shadow memory. If there are multiple inputs, the taint status of the output is logical OR of the taint status of all the inputs. Our approach is to insert a set of instructions around each instruction of the application code. The instrumented code updates the taint status in the shadow memory.

Performance Challenges

Although the conceptual idea of taint tracing is very simple, a similar approach described in [14] reports over 30 times slowdown when compared against native execution. Minimizing the overhead is a challenging task.

There are two major overheads. One is *instrumentation overhead* – the overhead caused by performing the instrumentation. The other is *tracing overhead* – the overhead incurred by executing the instrumented code to propagate taint status. We expect the tracing overhead to be much bigger than the instrumentation overhead, because instrumentation is only performed once for each application instruction, while the instrumented code may be executed many times. This is later verified by our experiments. Therefore, our optimization mainly emphasizes on the tracing overhead, i.e., reducing the number of instrumented instructions for each application instruction.

The tracing overhead can be divided into 3 parts described as follows.

1. **Shadow memory mapping.** The overhead for mapping application instruction operands to their shadow memory.

2. **Spill register.** For the usage of instrumented code, some general purpose registers need to be spilled (save and restore) in order to avoid disturbing the execution of the application code. In particular, if some instruction modifies the EFLAGs register, it should also be save and restored.

3. **Propagation overhead.** The overhead is incurred in order to propagate the taint information from source residence to destination residence.

```
l2 = l1[(addr >> 16) & 0xffff];
shadow = &l2[addr & 0xffff];
```

(a) C code for mapping addr to shadow.

```
01. mov addr → eax
02. sar eax, 10h → eax
03. and eax, 0ffffh → eax
04. mov [eax*4+l1] → eax
05. mov eax → l2
06. movzx word addr → eax
07. lea [eax*4+l2] → eax
08. mov eax → shadow
```

(b) Instructions generated by `gcc` for (a).

Figure 3: Shadow memory mapping with page-table like structure

Optimization

We apply optimization techniques to reduce all of the above tracing overheads. First, we realize that the mapping overhead of page-table like shadow memory structure used in the TaintCheck [14] paper is very large. As shown in Figure 3, it takes 8 instructions to find the shadow memory (shadow) for a application operand (addr). In order to reduce such mapping overhead, we devise a simple addressing mapping strategy: each shadow memory byte is mapped to the application memory byte by adding a constant offset, `shadow_base`. We have implemented a loader to realize such a mapping between the shadow memory and application memory. This requires no extra instructions to perform the mapping. Second, when instrumenting application instruction, we minimize register spilling with two techniques. We use dead register whenever possible. Also, we check if the application instruction will modify the EFLAGs. If so, we need not save and restore the EFLAGs, for example, arithmetic instructions. Third, we enforce byte-to-byte mapping from the application code to the shadow memory, which makes the taint propagation simple and very efficient.

3.1.3 Implementation

We implement our tool in Linux on x86 architecture, and use DynamoRIO to perform instrumentation. To simplify our work, we only instrument the general purpose instructions. We force the data addressable by the application and its shared library to a fixed specific memory space in order to enable our memory mapping strategy described above. We have implemented a loader to load the application and its shared library into a predefined memory space – between `0x000000000` and `0x57f00000`, which we call *application space*. We also intercept the `mmap` and `mmap2` system call to allocate the memory from the application space when the application requests more memory space.

Loader

We implement our loader by modifying the source code of Valgrind 2.4.0 [4]. It consists of two stages. In stage 1, it loads the code of stage 2 into the space between `0xb0000000` and `0xbfffffff`, which we call *monitor space*. Then the control is transferred to stage 2. In stage 2, it loads the application and its shared library into the application space, and loads DynamoRIO into the monitor space, then it transfers the control to DynamoRIO.

Next, DynamoRIO loads our shared library `dr-instrument.so` into the monitor space. This is used to perform the instrumentation and intercept system calls. Then DynamoRIO starts building and executing basic blocks of the application.

System Call Interception

We intercept system calls for several purposes: allocating shadow memory, marking taint status for data read from files or socket, and modifying temporary file operations. In Linux, system call is implemented by soft interrupt instruction `int80`. The system call ID and parameters are passed through the general purpose registers. When DynamoRIO builds basic blocks, the `int80` is identified. Our `dr-instrument.so` inserts instructions to call our functions `before_syscall` and `after_syscall` right before and after `int80`, respectively.

Function `before_syscall` checks the system call ID and its parameters. If the system call is `mmap` or `mmap2` for requesting memory, its parameters are modified to request memory in the application space. Function `after_syscall` examines the result returned from Linux, to check if the memory request is successful. If so, the corresponding shadow memory is also allocated and initialized.

Instrumentation

Another task of `dr-instrument.so` is to perform the instrumentation. Due to the complexity of x86 instruction, instrumentation is difficult and tedious. Valgrind [13] first translates x86 instruction to RISC like Ucode, performs transformation and instrumentation on Ucode, and then translates Ucode back to x86 instructions. In this way, it is easy for a user to perform code transformation. However, it cannot produce the optimal instrumented code. For performance reasons, we perform the instrumentation on the x86 instruction set directly using DynamoRIO. We mainly focus on the following two types of instructions for tracing the taint data propagation: data movement instructions like `mov`, `push`, and `cmov` (conditional move), and arithmetic instructions like `add` and `sar` (arithmetic right shift).

The instrumented instructions for each original instruction perform the following tasks in sequence: (1) spill a few registers for storing taint status, (2) map original operand (register or memory address) to its shadow memory, (3) load the taint status from shadow memory into the spilled register, (4) store the status into destination's shadow memory, and (5) restore the spilled registers. If the instrumented instructions modify the EFLAGs register, we need extra instructions to save and restore it. As described in subsection 3.1.2, our optimizations can reduce the number of instructions for (1), (2) and (5) to zero in most occasions. Also, the byte-to-byte mapping between application memory and shadow memory simplifies (3) and (4) greatly. Figure 4 and Figure 5 show two examples of our instrumentation for load and store, respectively. Here the `shadow_base` is 0x57f00000.

```
▷ load data from [edx+0x8] to eax
   mov [edx+0x8] → eax
```
(a) Before instrumentation.

```
▷ load taint status from shadow memory of [edx+0x8]
   mov [edx+0x57f00008] → eax
▷ store taint status into shadow memory of eax
   mov eax → [0xb11f1460]
▷ original instruction
   mov [edx+0x8] → eax
```
(b) After instrumentation.

Figure 4: Instrumentation for load

```
▷ store data from eax to [edx+0x8]
   mov eax → [edx+0x8]
```
(a) Before instrumentation.

```
▷ original instruction
   mov eax → [edx+0x8]
▷ load taint status from shadow memory of eax
   mov [0xb11f1460] → eax
▷ store taint status to shadow memory of [edx+0x8]
   mov eax → [edx+0x57f00008]
▷ restore eax
   mov [edx+0x8] → eax
```
(b) After instrumentation.

Figure 5: Instrumentation for store

3.1.4 Taint Checking

We implemented four types of taint checking. First, for the `printf` routine family, we check if the format string argument is tainted or not. Second, we check the taint status of the jump target, such as return address or function pointer, which is similar to program shepherding [12]. Third, we check whether the data storing the jump target is tainted. Finally, we check if some critical variable (for example, at control flow decision point) specified by user is tainted or not.

3.1.5 Discussion

There is a trade-off between the efficiency of shadow memory mapping and the usage of memory space. Our design reduces the number of instrumented code with the cost of doubled memory usage. Using a page-table like shadow memory structure as TaintCheck [14] can keep the shadow memory usage minimal, because it only allocates shadow memory for tainted data. Therefore, our tool is more suitable for large servers with plenty of memory and the application performance is concerned. While TaintCheck can be used for analysis purpose on small machines.

In addition, we realize that with our design the shadow memory becomes a critical area that must be protected from malicious access. Our shadow memory mapping strategy can prevent attackers from directly modifying the shadow memory as long as attackers' code is under the control of DynamoRIO. The reason is simple: when attackers issue an instruction to access an address `addr` in shadow memory area, the instrumented code will access memory at `addr+shadow_base`, which is beyond the boundary of the shadow memory area, and consequently will cause an invalid memory access exception.

3.2 Encrypted Session Data

DOG uses the Advanced Encryption Standard (AES) with a 256-bit key to protect against unauthorized access on private session data. The goal of our system is not in providing an Encrypted File System such as Microsoft's EFS [2] but merely to enforce the privacy of session data. Hence, the key is generated with a pseudo-random generator initialized with the timestamp of the application execution. DOG has a `Re-mapped Table` with entries keeping track of encryption and file pointer status. Encryption and decryption costs are kept low by dividing a file stream into fixed-size 16-bytes chunk as shown in Figure 6. Non-sequential writes and reads trigger encryption and decryption of only the overlapping chunks. In addition, DOG has a `File Write-out Alert Table` that lists the file descriptors that have permission to write sensitive data to file. In this way, our system alerts upon suspicious information leak.

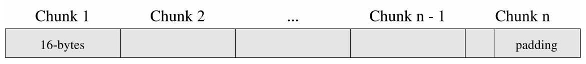


Figure 6: Format of encrypted file

4 Experimental Evaluation

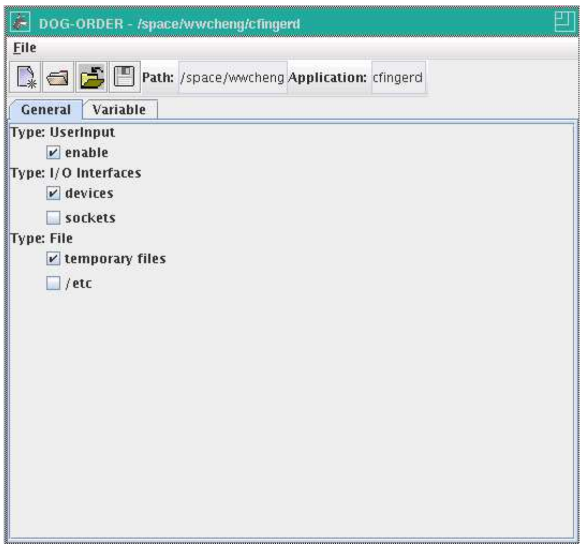
4.1 DOG

DOG-ORDER, shown in Figure 7 and 8, is the front-end for DOG. Only the system administrators can define TaintObjects for each application to be run under DOG. DOG-ORDER is written in Java version 1.5.0.

There are two types of TaintObjects: General and Variable, corresponding to coarse and fine-grained objects, respectively. Using the "General" tab, administrators can specify whether they want DOG to trace data entered originating from end-users, sockets, and specific interfaces, and also whether they want to encrypt temporary files and files in the /etc directory.

The "Variable" tab lets administrators view global variables of an application, if the application has been compiled with the symbol table. For each global variable, the administrator can specify whether DOG should trace copy propagation, whether DOG should trace arithmetic propagation, how DOG should respond when it observes a Format String or Buffer Overrun exploit, and finally whether DOG should encrypt the variable.

For each application, a configuration file is generated to communicate the policies and actions of TaintObjects to the backend. This file is vulnerable to attacks and our system protects its integrity, authenticity and confidentiality with the combination of symmetric- and asymmetric-key cryptography. The DOG backend has a set of RSA private



DOG-ORDER - /space/wwcheng/cfingerd

File

Path: /space/wwcheng Application: cfingerd

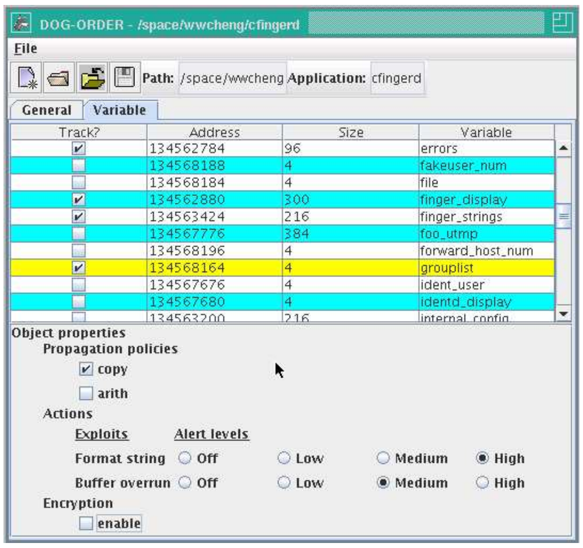
General | Variable

Type: UserInput
- [✓] enable

Type: I/O Interfaces
- [✓] devices
- [ ] sockets

Type: File
- [✓] temporary files
- [ ] /etc

Figure 7: DOG-ORDER General panel



DOG-ORDER - /space/wwcheng/cfingerd

File

Path: /space/wwcheng Application: cfingerd

General | Variable

Track?	Address	Size	Variable
✓	134562784	96	errors
	134568188	4	fakeuser_num
	134568184	4	file
✓	134562880	300	finger_display
✓	134563424	216	finger_strings
	134567776	384	foo_utmp
	134568196	4	forward_host_num
✓	134568164	4	grouplist
	134567676	4	ident_user
	134567680	4	identd_display
	134563200	216	internal_config

Object properties

Propagation policies
- [✓] copy
- [ ] arith

Actions

Exploits	Alert levels			
Format string	Off	Low	Medium	[✓ High]
Buffer overrun	Off	Low	[✓ Medium]	High

Encryption
- [ ] enable

Figure 8: DOG-ORDER Variable panel

and public keys (`backend_rsa_private_key`, `backend_rsa_public_key`). The DOG frontend also has a set of RSA private and public keys (`frontend_rsa_private_key`, `frontend_rsa_public_key`) as well as runtime generated AES private keys per configuration file (`aes_private_key`).

The frontend computes the SHA1 hash of the configuration file and generates a digital signature based on this hash using RSA with its private key. This allows the backend to verify the integrity and authenticity of the configuration file. The frontend then generates a random AES private key and encrypts the configuration file. To protect the privacy of the configuration file, it uses RSA with the backend's public key to encrypt the AES private key and the digital signature.

The backend uses its RSA private key to decrypt the AES private key and digital signature. Then it uses this AES private key to decrypt the configuration file and verifies that the computed hash is the same as the one signed by the digital signature. The steps are outlined in Figure 9.

4.2 Effectiveness of Taint Check

We evaluate our taint tracing tool with synthetic exploits on vulnerabilities of format string, buffer overflow, and critical variable.

Detecting format string attack

We wrote a program to test our taint tracing tool's ability to prevent format string attacks by detecting a tainted format string. This program accepts a user-supplied format string as the argument to `printf()`. This string has been maliciously devised to reveal some sensitive program data. When we ran this program with our taint tracing tool, DOG correctly detected that the `printf()` argument is tainted.

Detecting buffer overflow attack

In this test, we wrote a program that copies the input read from file to the local buffer. The input maliciously overflowed the local buffer such that some sensitive memory location – return address, is overwritten. In our case, the return address is modified to point to a sensitive function `grant_access()`. Our taint tracing tool can successfully detect that the return address is overwritten by tainted data when running the program under DOG.

In addition, our taint tracing tool can successfully detect all the buffer overflow attacks described in [18].

```
KEYS
At Frontend:
    set of fixed RSA public, private key
    AES private key (generated at runtime)
At Backend:
    set of fixed RSA public, private key

OPERATIONS
At Frontend:
    1. Ensuring integrity and authenticity
         h = SHA1(file)
         esig = RSAencrypt(frontend_rsa_private_key, h)
    2. Ensuring privacy
         generate aes_private_key
         efile = AESencrypt(aes_private_key, file)
         emsg = RSAencrypt(backend_rsa_public_key,
                           {esig, aes_private_key})
         Resulting information = {efile, emsg}
At Backend:
    {esig, aes_private_key} = RSAdecrypt(
                           backend_rsa_private_key, emsg)
    signed_hash = RSAdecrypt(frontend_rsa_public_key,
                           esig)
    decrypted_file = AESdecrypt(aes_private_key, efile)
    computed_hash = SHA1(decrypted_file)
    check(computed_hash == signed_hash)
```

Figure 9: Steps for configuration file protection.

Detecting critical variable attack

We also wrote a program in which a critical variable that determines the control flow of the program is overwritten by some maliciously devised input, causing the program to grant right to attackers. With our taint tracing tool, DOG can successfully detect when the critical variable is tainted.

Preventing temporary file attacks

We mimiked the information leakage vulnerability in Internet Message (IM) package in Redhat [1] where the name of a temporary file was easily guessed and the file could be corrupted by attacker. DOG successfully overcomes this problem as it ensures random naming on temporary files and these file mappings are hidden from the application. It is also able to prevent from temporary file race condition attacks since the generated file is encrypted and symbolic links are allowed.

4.3 Performance of Taint Tracing

In this experiment, we measured our taint tracing tool's performance with a subset of SPEC2000 INT. Our evaluation was performed on a system with 2.8GHz Pentium 4, 1024K L2 Cache, 1024MB of RAM, and 2048M swap, running Fedora Core 3.

We first profiled the time distribution of application running with our tracing tool. The result shows that more than 95% of time is spent in executing application code and instrumented code for almost all benchmarks. This justifies our previous claim that tracing overhead is dominant.

In order to evaluate the overhead of taint tracing, we ran our taint tracing tool with all security checks switched off. For each workload, we measured the running time of native execution, execution with our tracing tool, and execution with Valgrind Memcheck, which uses page-table like shadow memory and traces the status propagation in shadow memory. Figure 10 compares the relative slowdown between Valgrind Memcheck and our tracing tool under different workloads.

As can be observed from the figure, our taint tracing tool outperforms Valgrind Memcheck greatly with most workloads. The average relative slowdown of our tracing tool over all the workloads is 5.53. It is much smaller than Valgrind Memcheck's average slowdown, which is 29.62. We did not compare our tracing tool with TaintCheck [14] due to the lack of their source code. However, as reported in their paper, their performance is worse than Valgrind Memcheck.

With most workloads, the slowdown of our tracing tool is below 10. Twolf with test input is the exception because its native execution runs very fast, using only about 0.4s, in which case our tracing tool spent a large portion of time on initialization (about 80% according to our experiment). Under those workloads with ref input, the slowdown of our taint tracing tool is much smaller than those with test input. Further, most of workloads we tested are CPU bounded. We expect that the slowdown of our taint tracing tool would be even smaller with I/O bounded workloads.

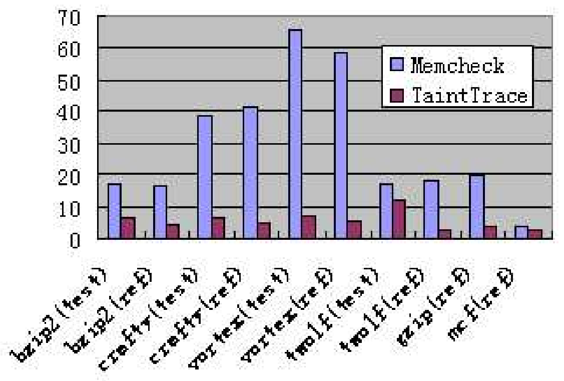


Figure 10: Relative slow down

5 Related Work

5.1 Program Monitoring

Program Shepherding [12] is a runtime monitoring system that keeps track of whether code has been modified since it was loaded, and checks each control transfer to check if the destination basic block has been modified or not. However, it cannot prevent any attacks that use existing-code.

5.2 Taint Tracing and Analysis

The most famous taint tracing work is Perl taint mode, which can prevent both obvious and subtle traps in code execution. While in taint mode, data from potentially untrusted sources, such as network sockets, is tagged as tainted. Perl is constantly and vigilantly checking to see if the script is going to do anything unsafe with the taint tags while the program runs.

More recent work on dynamic taint tracing and analysis includes [9, 17, 10, 14], which focus on various applications respectively. TaintBochs [9], Information Flow Traching [17] and Minos [10] all perform taint tracing at hardware level. TaintBochs [9], built on Bochs, the open source IA-32 simulator, is a tool based on whole-system simulation for analyzing how sensitive data are handled in large programs. The information flow tracking [17] project designs a hardware mechanism based on the SimpleScalar 3.0 tool set [6] for tracking information flow dynamically to protect programs against malicious attacks. It identifies spurious information flows and restricts the usage of spurious information. Minos [10], is a micro-architecture that implements Biba's low-water-mark integrity checking on individual words to detect attacks at runtime. It is also based on Bochs to check vulnerabilities at whole system level. A main limitation of the three systems is that they require specialized hardware, unless one wants to go through the trouble of building the custom hardware. Their performance deteriorate greatly compared with native execution when used with hardware emulators. In addition, due to their hardware design, it is difficult for users to configure their security requirements. Similar to ours, TaintCheck [14] developed a taint tracing based on code instrumenting, but is based on Valgrind [4]. As mentioned before, its page-table like structure contributes to its over 30 times slow down when compared against native execution.

There are also a variety of works developed for static taint analysis. [16] developed a static analysis system for automatically detecting format string bugs at compile-time. [7] uses tainting-style analysis to find security errors with system-specific extensions linked into the compiler written by programmers. Due to the lack of runtime information, most of them are overly conservative and inaccurate.

6 Conclusion and Future Work

In this paper, we present DOG, an efficient information flow tracing and program monitoring security system. Our system is able to protect against various kinds of attacks, such as format string attack and buffer overflow attacks. Our experiment demonstrated that our system is much more efficient and practical than other similar tools.

In the future, we will improve our system along the following two directions. First, we can further improve the performance of our taint tracing tool. Current taint tracing performs instrumentation without knowledge of any global information, and would produce some redundant code. By analyzing the data flow of basic block, we can find registers to steal with minimal spilling overhead. The information would also help reducing overhead on redundant taint propagation. Second, similar to backtracking [11], we can log the system call. If an intrusion is detected later, the logged information can help administrator analyze how the attack happens and discover the application's vulnerability.

References

[1] http://rhn.redhat.com/errata/RHSA-2003-039.html. Redhat Security Advisory, RHSA-2003:039-09, February, 2003.

[2] Encrypting File System in Windows XP and Windows Server, 2003. http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/cryptfs.mspx.

[3] Perl security manual page. http://www.perldoc.com/perl5.6/pod/perlsec.html.

[4] Valgrind. http://valgrind.org/.

[5] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC CPU2000 benchmark suite, 2000. http://www.spec.org/osg/cpu2000/.

[6] D. B. an T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[7] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[8] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, September 2004. http://www.cag.csail.mit.edu/rio/.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *13th USENIX Security Symposium*, 2004.

[10] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*, 2004.

[11] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, New York, NY, USA, 2003. ACM Press.

[12] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *11th USENIX Security Symposium*, 2002.

[13] N. Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, University of Cambridge, November 2004.

[14] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium*, 2005.

[15] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, 2005.

[16] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, 2001.

[17] G. E. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI*, 2004.

[18] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *10th Network and Distributed System Security Symposium*, 2003.