Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

James Newsome Dawn Song

May 2004
Last updated July 2005
CMU-CS-04-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1 Introduction

Software vulnerabilities such as buffer overruns and format string vulnerabilities have had a devastating effect on the Internet. Worms such as CodeRed and Slammer exploit software vulnerabilities and can compromise hundreds of thousands of hosts within hours or even minutes, and cause millions of dollars of damage [30, 48]. To successfully combat fast Internet worm attacks, we need automatic detection and defense mechanisms. First, we need automatic detection mechanisms that can detect new attacks for previously unknown vulnerabilities. A detection mechanism should be easy to deploy, result in few false positives and few false negatives, and detect attacks early, before a significant fraction of vulnerable systems are compromised. Second, once a new exploit attack is detected, we must quickly develop filters (a.k.a. attack signatures) that can be used to filter out attack packets efficiently, and hence protect vulnerable hosts from compromise until the vulnerability can be patched. Because a new worm can spread quickly, signature generation must be automatic—no manual intervention can respond quickly enough to prevent a large number of vulnerable hosts from being infected by a new fast-spreading worm.

We need fine-grained attack detectors for commodity software. Many approaches have been proposed to detect new attacks. These approaches roughly fall into two categories: *coarse-grained detectors*, that detect anomalous behavior, such as scanning or unusual activity at a certain port; and *fine-grained detectors*, that detect attacks on a program's vulnerabilities. Coarse-grained detectors may result in frequent false positives, and do not provide detailed information about the vulnerability and how it is exploited. Thus, it is desirable to develop fine-grained detectors that produce fewer false positives, and provide detailed information about the vulnerability and exploit.

Several approaches for fine-grained detectors have been proposed that detect when a program is exploited. Most of these previous mechanisms require source code or special recompilation of the program, such as StackGuard [19], PointGuard [18], full-bounds check [24, 42], LibsafePlus [8], FormatGuard [17], and CCured [32]. Some of them also require recompiling the libraries [24, 42], or modifying the original source code, or are not compatible with some programs [32, 18]. These constraints hinder the deployment and applicability of these methods, especially for commodity software, because source code or specially recompiled binaries are often unavailable, and the additional work required (such as recompiling the libraries and modifying the original source code) makes it inconvenient to apply these methods to a broad range of applications. Note that most of the large-scale worm attacks to date are attacks on commodity software.

Thus, it is important to design fine-grained detectors that work on commodity software, *i.e.*, work on arbitrary binaries without requiring source code or specially recompiled binaries. This goal is difficult to achieve because important information, such as type information, is not generally available in binaries. As a result, existing exploit detection mechanisms that do not use source code or specially compiled binary programs, such as LibSafe [9], LibFormat [41], Program Shepherding [27], and the Nethercote-Fitzhardinge bounds check [33], are typically tailored for narrow types of attacks and fail to detect many important types of common attacks (see Section 7 for details).

We need automatic tools for exploit analysis and signature generation. Because fine-grained detectors are expensive and may not be deployed on every vulnerable host, once a new exploit attack is detected, it is desirable to generate faster filters that can be widely deployed to filter out exploit requests before they reach vulnerable hosts/programs. One important mechanism is content-based filtering, where content-based signatures are used to pattern-match packet payloads to determine whether they are a particular attack. Content-based filtering is widely used in intrusion detection systems such as Snort [37], Bro [36], and Cisco's NBAR system [50], and has been shown to be more effective than other mechanisms, such as source-based filtering for worm quarantine [31]. However, these systems all use *manually generated* databases of signatures. Manual signature generation is clearly too slow to react to a worm that infects hundreds of thousands of machines in a matter of hours or minutes. We need to have automatic exploit analysis and signature generation to quickly generate signatures for attack filtering after an exploit attack has been detected.

Our contributions. In this paper, we propose a new approach, *dynamic taint analysis*, for the automatic detection, analysis, and signature generation of exploits on commodity software. In dynamic taint analysis, we label data originating from or arithmetically derived from untrusted sources such as the network as *tainted*. We keep track of the propagation of tainted data as the program executes (*i.e.*, what data in memory is tainted), and detect when tainted data is used in dangerous ways that could indicate an attack. This approach allows us to detect *overwrite attacks*, attacks that cause a sensitive value (such as return addresses, function pointers, format strings, *etc.*) to be overwritten with the attacker's data. Most commonly occurring exploits fall into this class of attacks. After an attack has been detected, our dynamic taint analysis can automatically provide information about the vulnerability, how the vulnerability was exploited, and which part of the payload led to the exploit of the vulnerability. We show how this information could be used to automatically generate signatures for attack filtering. We have developed an automatic tool, *TaintCheck*, to demonstrate our dynamic taint analysis approach. TaintCheck offers several unique benefits:

- **Does not require source code or specially compiled binaries.** TaintCheck operates on a normally compiled binary program. This makes TaintCheck simple and practical to use for a wide variety of programs, including proprietary programs and commodity programs for which no source code is available.

- **Reliably detects most overwrite attacks.** TaintCheck's default policy detects format string attacks, and overwrite attacks that attempt to modify a pointer used as a return address, function pointer, or function pointer offset. Its policy can also be extended to detect other overwrite attacks, such as those that attempt to overwrite data used in system calls or security-sensitive variables.

- **Has no known false positives.** In our experiments, TaintCheck gave no false positives in its default configuration. As we discuss in Section 3, in many cases when a false positive could occur, it is a symptom of a potentially exploitable bug in the monitored program. For programs where the default policy of TaintCheck could generate a false positive, we show in Section 3 that it is straightforward to configure TaintCheck to reduce or eliminate those false positives.

- **Enables automatic semantic analysis based signature generation.**

  We propose a new approach for automatic signature generation: using *automatic semantic analysis* of attack payloads to identify which parts of the payload could be useful in an attack signature. Previous work in automatic signature generation uses content pattern extraction to generate signatures [26, 29, 46]. The information provided by semantic analysis could be used to generate a signature directly, or as hints to content pattern extraction techniques. Because semantic analysis provides information about the vulnerability and how it is exploited, it could potentially allow an accurate signature to be automatically generated using fewer payloads than would be necessary using content pattern extraction alone. By requiring fewer attack payloads, semantic analysis could generate a signature at an earlier stage of a worm epidemic, thus minimizing damage caused by a new worm.

TaintCheck could be used to perform automatic semantic analysis of attack payloads, because it monitors how each byte of each attack payload is used by the vulnerable program at the processor-instruction level. As a first step, we show that TaintCheck can be used to identify the value used to overwrite a return address or function pointer. The most significant bytes of this value can be used as part of a signature. We also show that for text-based protocols such as HTTP, it can be used as a signature by itself, with only a small false positive rate.

Moreover, we show how TaintCheck can be used as an accurate classifier both in existing automatic signature generation systems, and in an automatic semantic analysis signature generation system. As an accurate classifier, TaintCheck can be used to accurately identify new attacks. It can also be used to verify the quality of generated signatures by determining whether requests that match a new signature actually contain an attack.

TaintCheck adds a new point in the design space of automatic detection and defense, and is the first approach that achieves all the above properties.

The current implementation of TaintCheck slows server execution between 1.5 and 40 times. However, our prototype has not been optimized. Several techniques described in Section 4 will lead to a more efficient implementation. Additionally, we show in Section 5 that monitoring even a small fraction of incoming requests with TaintCheck could help to detect a new worm in the early stages of an epidemic.

The rest of the paper is organized as follows. We describe TaintCheck's design and implementation, and how it detects various attacks, in Section 2. We show TaintCheck is able to detect a wide variety of attacks with few false positives and negatives in Section 3. We evaluate the effectiveness and performance of TaintCheck in Section 4. We discuss how TaintCheck can be applied to detection of new attacks in Section 5, and to automatic signature generation in Section 6. We present related work in Section 7, and our conclusions in Section 8.

2 TaintCheck design and implementation

TaintCheck is a novel mechanism that uses *dynamic taint analysis* to detect when a vulnerability such as a buffer overrun or format string vulnerability is exploited. We first give an overview of our dynamic taint analysis approach, and then describe how we use this approach in the design and implementation of TaintCheck.

Dynamic taint analysis Our technique is based on the observation that in order for an attacker to change the execution of a program illegitimately, he must cause a value that is normally derived from a trusted source to instead be derived from his own input. For example, values such as jump addresses and format strings should usually be supplied by the code itself, not from external untrusted inputs. However, an attacker may attempt to exploit a program by overwriting these values with his own data.

We refer to data that originates or is derived arithmetically from an untrusted input as being *tainted*. In our dynamic taint analysis, we first mark input data from untrusted sources tainted, then monitor program execution to track how the tainted attribute propagates (*i.e.*, what other data becomes tainted) and to check when tainted data is used in dangerous ways. For example, use of tainted data as jump addresses or format strings often indicates an exploit of a vulnerability such as a buffer overrun or format string vulnerability.

Note that our approach detects attacks at the time of *use*, *i.e.*, when tainted data is used in dangerous ways. This significantly differs from many previous approaches which attempt to detect when a certain part of memory is illegitimately overwritten by an attacker at the time of the *write*. It is not always possible at the time of a write to detect that the overwrite is illegitimate, especially for approaches not using source code or specially recompiled binaries. In contrast, our approach does not rely on detection at the time of overwrite and is independent of the overwriting method, and thus can detect a wide range of attacks.

Design and implementation overview TaintCheck performs dynamic taint analysis on a program by running the program in its own emulation environment. This allows TaintCheck to monitor and control the program's execution at a fine-grained level. Specifically, we implemented TaintCheck using Valgrind [34]. Valgrind is an open source x86 emulator that supports extensions, called *skins*, which can instrument a program as it is

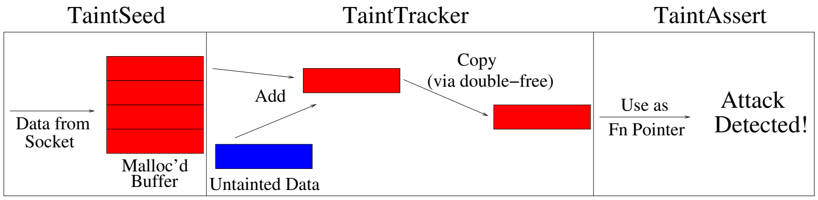


Figure 1: TaintCheck detection of an attack. (Exploit Analyzer not shown).

run.[1] We also have a Windows implementation of TaintCheck that uses DynamoRIO [3], another dynamic binary instrumentation tool. For the remainder of this paper, we refer to the Valgrind implementation unless otherwise specified.

Whenever program control reaches a new basic block, Valgrind first translates the block of x86 instructions into its own RISC-like instruction set, called *UCode*. It then passes the UCode block to TaintCheck, which instruments the UCode block to incorporate its taint analysis code. TaintCheck then passes the rewritten UCode block back to Valgrind, which translates the block back to x86 code so that it may be executed. Once a block has been instrumented, it is kept in Valgrind's cache so that it does not need to be reinstrumented every time it is executed.

To use dynamic taint analysis for attack detection, we need to answer three questions: (1) What inputs should be tainted? (2) How should the taint attribute propagate? (3) What usage of tainted data should raise an alarm as an attack? To make TaintCheck flexible and extensible, we have designed three components: *TaintSeed, TaintTracker,* and *TaintAssert* to address each of these three questions in turn. Figure 1 shows how these three components work together to track the flow of tainted data and detect an attack. Each component has a default policy and can easily incorporate user-defined policies as well. In addition, each component can be configured to log information about taint propagation, which can be used by the fourth component we have designed, the *Exploit Analyzer*. When an attack is detected, the Exploit Analyzer performs post-analysis to provide information

[1] Note that while Memcheck, a commonly used Valgrind extension, is able to assist in debugging memory errors, it is not designed to detect attacks. It can detect some conditions relevant to vulnerabilities and attacks, such as when unallocated memory is used, when memory is freed twice, and when a memory write passes the boundary of a `malloc`-allocated block. However, it does not detect other attacks, such as overflows within an area allocated by one `malloc` call (such as a buffer field of a struct), format string attacks, or stack-allocated buffer overruns.

about the attack, including identifying the input that led to the attack, and semantic information about the attack payload. This information can be used in automatic attack signature generation, as we show in Section 6.

2.1 TaintSeed

TaintSeed marks any data that comes from an untrusted source of input as tainted. By default, TaintSeed considers input from network sockets to be untrusted, since for most programs the network is the most likely vector of attack. TaintSeed can also be configured to taint inputs from other sources considered untrusted by an extended policy, *e.g.*, input data from certain files or stdin.

Each byte of memory, including the registers, stack, heap, *etc.*, has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted, or a NULL pointer if it is not. We use a page-table-like structure to ensure that the shadow memory uses very little memory in practice. TaintSeed examines the arguments and results of each system call, and determines whether any memory written by the system call should be marked as tainted or untainted according to the TaintSeed policy. When the memory is tainted, TaintSeed allocates a Taint data structure that records the system call number, a snapshot of the current stack, and a copy of the data that was written. The shadow memory location is then set to a pointer to this structure. This information can later be used by the Exploit Analyzer when an attack is detected. Optionally, logging can be disabled, and the shadow memory locations can simply store a single bit indicating whether the corresponding memory is tainted.

2.2 TaintTracker

TaintTracker tracks each instruction that manipulates data in order to determine whether the result is tainted. UCode Instructions fall into three categories: *data movement instructions* that move data (LOAD, STORE, MOVE, PUSH, POP, *etc.*), *arithmetic instructions* that perform arithmetic operations on data (ADD, SUB, XOR, *etc.*), and those that do neither (NOP, JMP, *etc.*). The default policy of TaintTracker is as follows: for data movement instructions, the data at the destination will be tainted if and only if any byte of the data at the source location is tainted; for arithmetic instructions, the result will be tainted if and only if any byte of the operands is tainted. While arithmetic instructions also affect the processor's condition flags, we do not track whether the flags are tainted, because it is normal for untrusted data to influence them. Note that for both data movement and arithmetic instructions, literal values are considered untainted, since they originate either from the source code of the program or from the compiler.

A special case is for constant functions where the output of the function does not depend on the inputs. For example, a common IA-32 idiom to zero out a register, "`xor eax, eax`", always sets `eax` to be zero regardless of whether the original value in `eax` is tainted or not. TaintTracker recognizes these special cases such as `xor eax, eax` and `sub eax, eax`, and sets the result location to be untainted. Note that there can be more general cases of constant functions where a sequence of instructions computes a constant function. We do not handle these more general cases. However, such cases will only make the dynamic taint analysis conservative and it has not been an issue in practice.

In Section 3 we show how tainted data is sometimes translated via a lookup table. For example, an implementation of `toupper`, which converts a string to all upper case, is to use each byte as an index into a translation table. TaintTracker optionally taints data that is loaded via a tainted pointer, so that a tainted string translated in this manner remains tainted.

In order to track the propagation of tainted data, TaintTracker adds instrumentation before each data movement or arithmetic instruction. When the result of an instruction is tainted by one of the operands, TaintTracker sets the shadow memory of the result to point to the same Taint structure as the tainted operand. Optionally, TaintTracker can instead allocate a new Taint structure with information about the relevant instruction (including the operand locations and values, and a snapshot of the stack) that points back to the previous Taint structure. When an attack is detected, the Exploit Analyzer can follow this chain of Taint structures backwards to determine how the tainted data propagated through memory.

2.3 TaintAssert

TaintAssert checks whether tainted data is used in ways that its policy defines as illegitimate. TaintAssert's default policy is designed to detect format string attacks, and attacks that alter jump targets including return addresses, function pointers, or function pointer offsets. When TaintCheck detects that tainted data has been used in an illegitimate way, signalling a likely attack, it invokes the Exploit Analyzer to further analyze the attack.

The following are potentially illegitimate ways in which tainted data might be used. TaintAssert's policy can be specified to check for any of these independently.

- **Jump addresses** By default, TaintAssert checks whether tainted data is used as a jump target, such as a return address, function pointer, or function pointer offset. Many attacks attempt to overwrite one of these in order to redirect control flow either to the attacker's code, to a standard library function such as `exec`, or to another point in the program (possibly circumventing security checks). In contrast, there are very few scenarios in which tainted data would be used as a jump target during normal usage of a program, and we have not found any such examples in our testing. Hence, these checks detect a wide variety of attacks while generating very few false positives.

  Note that jump tables are a possible exception to this rule. A jump table could use user input as an offset to a jump address. This is an acceptable programming practice if there are checks in place to sanitize the tainted data. gcc does not appear to construct jump tables in this way in practice, but other compilers or hand-coded assembly might. See Section 3 for further discussion of this scenario.

  We implemented these checks by having TaintCheck place instrumentation before each UCode `jump` instruction to ensure that the data specifying the jump target is not tainted. Note that IA-32 instructions that have jump-like behavior (including `call` and `ret`) are translated into UCode jump instructions by Valgrind.

- **Format strings** By default, TaintAssert also checks whether tainted data is used as a format string argument to the `printf` family of standard library functions. These checks detect format string attacks, in which an attacker provides a malicious format string to trick the program into leaking data or into writing an attacker-chosen value to an attacker-chosen memory address. These checks currently detect whenever tainted data is used as a format string, even if it does not contain malicious format specifiers for attacks. This could be used to discover previously unknown format string vulnerabilities. Optionally, TaintAssert can instead only signal when the format string both is tainted and contains dangerous format specifiers such as `%n`. This option is useful when a vulnerability is already known, and the user only wants to detect actual attacks.

  To implement these checks, we intercept calls to the `printf` family of functions (including `syslog`) with wrappers that request TaintCheck to ensure that the format string is not tainted, and then call the original function. For most programs, this will catch any format string attack and not interfere with normal functionality. However, if an application uses its own implementation of these functions, our wrappers may not be called.

- **System call arguments** TaintAssert can check whether particular arguments to particular system calls are tainted, though this is not enabled in TaintCheck's default policy. This could be used to detect attacks that overwrite data that is later used as an argument to a system call. These checks are implemented using Valgrind's callback mechanism to examine the arguments to each system call before it is made.

  As an example, we implemented an optional policy to check whether the argument specified in any `execve` system call is tainted. This could be used to detect if an attacker attempts to overwrite data that is later used to specify the program to be loaded via an `execve` system call. We disabled this check by default, because some programs use tainted data in this way during normal usage. A notable example is that Apache uses part of a URL string as the argument to `execve` when a CGI is requested.

- **Application or library-specific checks** TaintAssert can also be configured to detect attacks that are specific to an application or library. It can do this by checking specified memory ranges at specified points of the program. In particular, it can be configured to check whether a particular argument to a particular function is tainted whenever that function is called. An example of this is checking the format strings supplied to `printf`-style functions, as described above.

  To implement this, TaintCheck could be told to check whether a particular address range or register is tainted whenever the program counter reaches a particular value, or whenever it is used in a certain way. The address range specified could be absolute, or could be relative to the current stack frame. This policy is application dependent and is disabled by default.

These checks are sufficient to catch a wide range of attacks. There are two other types of checks we also considered, but decided not to use. The first is tracking which flags are tainted, and checking when a tainted flag is used to alter control flow. This could detect when the attacker overwrites a variable that affects the behavior of the program. However, tainted data is used to alter control flow on a regular basis, and it is unclear whether there is a reliable way to differentiate the normal case from an attack.

The second type is checking whether addresses used in data movement instructions are tainted. This could detect when an attacker overwrites a data pointer in order to control where data is moved to or loaded from. However, it is common to use tainted data as an offset to data movement instructions, particularly in the case of arrays.

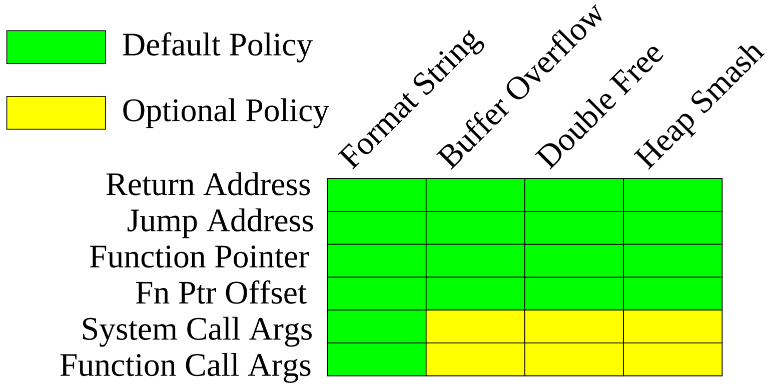
2.4 Exploit Analyzer

When TaintAssert detects that tainted data has been used in a way violating its security policy, thus signaling a likely exploit, the Exploit Analyzer can provide useful information about how the exploit happened, and what the exploit attempts to do. These functions are useful for identifying vulnerabilities and for generating exploit signatures.

Information logged by TaintSeed and TaintTracker shows the relevant part of the execution path in between tainted data's entry into the system, and its use in an exploit. By backtracing the chain of Taint structures, the Exploit Analyzer provides information including the original input buffer that the tainted data came from, the program counter and call stack at every point the program operated on the relevant tainted data, and at what point the exploit actually occurred. The Exploit analyzer can use this information to help determine the nature and location of a vulnerability quickly, and to identify the exploit being used.

The Exploit Analyzer can optionally allow an attack to continue in a constrained environment after it is detected. We currently implement an option to redirect all outgoing connections to a logging process. This could be used to collect additional samples of a worm, which can be used to help generate a signature for that worm.

The Exploit Analyzer could also be used to provide semantic information about the attack payload. This information can be used to automatically generate attack signatures more accurately and with fewer samples than is possible with purely content-based analysis of the attack payload. To demonstrate this idea, the Exploit Analyzer currently identifies the value used to overwrite the return address. We show in Section 6 that the most significant bytes of this value can be used in a signature of the attack. Note that our techniques are related to dynamic program slicing [7, 28], although dynamic program slicing considers control-flow dependencies and is often based on source code analysis.



🟩 Default Policy

🟨 Optional Policy

	Format String	Buffer Overflow	Double Free	Heap Smash
Return Address	🟩	🟩	🟩	🟩
Jump Address	🟩	🟩	🟩	🟩
Function Pointer	🟩	🟩	🟩	🟩
Fn Ptr Offset	🟩	🟩	🟩	🟩
System Call Args	🟩	🟨	🟨	🟨
Function Call Args	🟩	🟨	🟨	🟨

Figure 2: Attack detection coverage.

3 Security analysis of TaintCheck

In this section, we analyze the attacks that can be detected by TaintCheck and the false positives and false negatives incurred by TaintCheck.

Attacks detected by TaintCheck Figure 2 classifies overwrite attacks by the type of value that is overwritten, and by the method used to perform the overwrite. In general, TaintCheck is capable of detecting any overwrite attack that overwrites a value that would not normally be tainted. TaintCheck's default policy is that jump targets and format strings should not be tainted, allowing it to detect attacks that overwrite jump targets (such as return addresses, function pointers, and function pointer offsets), whether altered to point to existing code (existing code attack) or injected code (code injection attack), and all format string attacks. It's important to note that most of the worm attacks we have seen to date fall into these categories, including all the major worms such as the Slammer Worm and the CodeRed Worm. TaintCheck's policy can also be customized in order to detect an even wider range of attacks, as described in Section 2.3

False negative analysis A false negative occurs if an attacker can cause sensitive data to take on a value of his choosing without that data becoming tainted. This can be achieved if the altered data does not originate and is not arithmetically derived from untrusted inputs, but is still influenced by untrusted inputs. In particular, because we do not consider the tainted attribute of flags, the attacker can alter data by influencing the control flow of conditional branches to evade tainting. For example, suppose the variable

x is tainted. A structure of the form `if (x == 0) y = 0; else if (x == 1) y = 1; ...` is semantically the same as $y = x$ but would not cause y to become tainted, since the value for y is only influenced indirectly by x, via the condition flags. If the attacker could later cause y to overwrite a sensitive value, the attack would be undetected.

Another potential problem is if tainted data is used as an index into a table. For example, IIS translates ASCII input into Unicode via a table [20]. The resulting translation is not tainted, because the values were copied from hard-coded literals, rather than arithmetically derived from the input. We have added an option to taint data that is loaded via tainted pointers to handle this case.

Other false negatives can occur if TaintCheck is configured to trust inputs that should not be trusted. The current default configuration of not trusting data read from network sockets is sufficient to detect most remote attacks. However, an attacker may be able to control data from other input sources, depending on the application. An example of this is a vulnerability in the innd news server, in which data from the network is first written to a file on disk, and then read back into memory [4, 20]. These types of false negatives can be minimized by using a more restrictive policy of what inputs should be tainted. In our experiments, marking data read from files other than dynamically loaded libraries did not cause false positives, except in the case of some configuration files. In those cases, it is straightforward to configure TaintCheck not to taint data read from those files.

Analysis and handling of false positives In cases where TaintCheck detects that tainted data is being used in an illegitimate way even when there is no attack taking place, it can mean one of two things. First, it could mean that the program contains a vulnerability that should be fixed. For example, the program may be using an unchecked input as a format string. In this case, the best solution is to fix the vulnerability, possibly using TaintCheck's Exploit Analyzer to help understand it. Another possibility is to configure TaintCheck to only signal an attack if some other condition is also true- for example, if a tainted format string actually does contain dangerous format specifiers (such as %n).

The other possibility is that the program performs sanity checks on the tainted data before it is used, ensuring that the operation is actually safe. For example, the program might use tainted data as a format string, but only after it has ensured that it does not contain any malicious format specifiers such as %n (which would signify a possible format string attack). Another example is that a program could use tainted data as a jump target in a jump table, after checking that it is within expected bounds. Fortunately, these cases occur relatively rarely and usually at fixed places (program counters) in programs. Most of these false positives can be detected by running programs on legitimate inputs through a training phase. In these cases, TaintCheck can either be configured to ignore the particular failed taint assertion, or, if additional information is available, to *untaint* the data immediately after it has been sanity checked. The latter option is safer, since an attacker may attempt to overwrite the data again after it has been sanity checked.

4 Evaluation

We evaluate TaintCheck's compatibility and incidence of false positives in Section 4.1, its effectiveness against various attacks in Section 4.2, and its performance in Section 4.3.

4.1 Compatibility and false positives

We used TaintCheck to monitor a number of programs in order to check for false positives, and to verify that the programs were able to run normally. We tested several server programs: apache, ATPhttpd, bftpd, cfingerd, and named; client programs: ssh and firebird; and non-network programs: gcc, ls, bzip2, make, latex, vim, emacs, and bash.

All of these programs functioned normally when run under TaintCheck, and no false positives occurred using TaintCheck's default policy of tainting data from network sockets and asserting that jump targets and format strings are untainted. In our evaluation using named, we replayed a trace containing 158,855 DNS queries to the primary nameserver at Princeton University. This nameserver is also the secondary server for a top level European domain, and handles outbound queries for Princeton users. Hence, this trace contains a diverse set of requests from a diverse set of clients. Our named server was configured to resolve each request by performing a recursive query. The TaintCheck-monitored named server behaved correctly and did not generate any false positives.

To further test for false positives, we tried running all of the client programs and non-network programs with a policy to taint data read from standard input, and data read from files (except for files owned by root, notably including dynamically loaded libraries). The only additional false positives that resulted were in vim and firebird. In both cases, the program appears to be using data read from one of its respective configuration files

Program	Overwrite Method	Overwrite Target	Detected
ATPhttpd	buffer overflow	return address	✔
synthetic	buffer overflow	function pointer	✔
synthetic	buffer overflow	format string	✔
synthetic	format string	none (info leak)	✔
cfingerd	`syslog` format string	GOT entry	✔
wu-ftpd	`vsnprintf` format string	return address	✔

Table 1: Evaluation of TaintCheck's ability to detect exploits

as an offset to a jump address. This could easily be fixed by configuring TaintCheck to trust the corresponding configuration files.

4.2 Evaluation of attack detection

We tested TaintCheck's ability to detect several types of attacks, including several synthetic and actual exploits. Most of these attacks attempted to use a vulnerability to overwrite a sensitive value. The one exception is an information leak attack in which a user-supplied format string contained format specifiers, causing the program to output data from the stack. As Table 1 shows, TaintCheck successfully detected each attack. For the format string attacks that attempted to overwrite another value, TaintCheck detected both that a tainted format string was being used, and that the other value had been overwritten. Additionally, TaintCheck successfully identified the value used to overwrite the return address in the ATPhttpd exploit. We show in Section 6 how this can be useful when generating a signature for buffer overflow attacks.

4.2.1 Synthetic exploits

In this section, we evaluate TaintCheck using synthetic exploits on buffer overruns that overwrite return addresses, function pointers, and format string vulnerabilities. In all these evaluations, TaintCheck successfully detected all attacks and resulted in no false positives.

Detecting overwritten return address In order to test TaintCheck's ability to detect an overwritten return address, we wrote a small program with a buffer overflow vulnerability. The program uses the dangerous "gets" function in order to get user input. An input that is too long will overflow the buffer and begin overwriting the stack. We performed a test in which the return address is overwritten with the address of an existing function in the code. TaintCheck was able to detect the attack because the return address was tainted from user input.

Detecting overwritten function pointer In a similar test, we verified TaintCheck's ability to detect an overwritten function pointer. We wrote a program with a stack buffer overflow vulnerability where the overrun buffer could overwrite a function pointer on the stack. Again, TaintCheck correctly detected the attack because the function pointer was tainted by user input during the buffer overrun.

Detecting format string vulnerability Finally, we wrote another program to verify TaintCheck's ability to detect a tainted format string, which can lead to a format string attack. This program took a line of input from the user, and printed it back by using it as the format string in a call to `printf`. When we ran this program under TaintCheck, TaintCheck correctly detected that a tainted format string was being used in `printf`. As a further test, we wrote a program with a buffer overrun vulnerability that allowed the attacker to overwrite a format string. An attacker might choose to overwrite the format string to perform a format string attack instead of directly overwriting the return address in order to evade some buffer-overflow protection mechanisms. Again, we found that TaintCheck was able to determine correctly when the format string was tainted.

4.2.2 Actual exploits

In this section, we evaluate TaintCheck on exploits to three vulnerable servers: a web server, a finger daemon, and an FTP server. In all these evaluations, TaintCheck successfully detected all the attacks and incurred no false positives during normal program execution.

ATPhttpd exploit ATPhttpd [40] is a web server program. Versions 0.4b and lower are vulnerable to several buffer overflow vulnerabilities. We obtained an exploit that sends the server a malicious GET request [39]. The request asks for a very long filename, which is actually shellcode and a return address. The filename overruns a buffer, causing the return address to be overwritten. When the function attempts to return, it jumps instead to the shellcode inside the file name. The attacker is then given a remote shell.

TaintCheck correctly detected that the return address was tainted when the server was attacked, and did not generate any false positives when serving normal GET requests. TaintCheck also correctly identifies the return address value that overwrites the previous value. As we show in Section 6, this can sometimes be used as a signature for an attack.

cfingerd exploit cfingerd is a finger daemon that contains a format string vulnerability in versions 1.4.2 and lower. We obtained an exploit for this vulnerability that works as follows. When cfingerd prompts for a user name, the exploit responds with a string beginning with "version", and also containing malicious code. Due to another bug, cfingerd copies the whole string into memory, but only reads to the end of the string "version". Thus, the malicious code is allowed to reside in memory, and the string appears to be a legitimate query.

cfingerd later contacts the identd daemon running on the querier's machine. The exploit runs its own identd, responding with a string that will be later used as a format string to the `syslog` function. When cfingerd uses this format string, the entry for the `exit` function in the global offset table is overwritten to point to the malicious code that was inserted in the first step. When cfingerd finishes processing the query, it attempts to exit, but is caused to execute the attacker's code instead.

During normal usage, TaintCheck correctly detects that tainted data is being used as a format string. When we used the exploit, TaintCheck detected the tainted format string, and later detected when the program attempted to use the tainted pointer in the global offset table.

wu-ftpd exploit Version 2.6.0 of wu-ftpd has a format string vulnerability in a call to `vsnprintf`. We obtained an exploit for this vulnerability that uses the format string to overwrite a return address. TaintCheck successfully detects both that the format string supplied to `vsnprintf` is tainted, and that the overwritten return address is tainted.

Slapper The Slapper worm [1] exploits a heap buffer overrun vulnerability in Apache's mod_ssh module. This is used to corrupt the heap metadata, which later cause's the GOT entry of `free` to be overwritten with a pointer to the worm's code. TaintCheck successfully detects that the GOT entry is tainted when it is used.

Slammer The Slammer worm [2] exploits a stack buffer overrun vulnerability in MS-SQL Server to overwrite a return address. The Windows implementation of TaintCheck successfully detects this attack.

4.3 Performance evaluation

We measured TaintCheck's performance using two "worst-case" workloads (a CPU-bound workload and a short-lived process workload), and what we consider to be a more common workload (a long-lived I/O-bound workload). For each workload, we measured the performance when the program was run natively, when it ran under Nullgrind (a Valgrind skin that does nothing), when it ran under Memcheck (a commonly used Valgrind skin that checks for run-time memory errors, such as use of uninitialized values), and when it ran under TaintCheck. Our evaluation was performed on a system with a 2.00 GHz Pentium 4, and 512 MB of RAM, running RedHat 8.0.

CPU-bound: bzip2 In order to evaluate the penalty from the additional instrumentation that TaintCheck writes into the monitored binary at run-time, we evaluated the performance of bzip2, a CPU-bound program. Specifically, we measured how long bzip2 took to compress a 15 MB package of source code (Vim 6.2). When run normally, the compression took 8.2 seconds to complete. When run under Valgrind's Nullgrind skin, the task took 25.6 seconds (3.1 times longer). When using MemCheck, it took 109 seconds (13.3 times longer). When using TaintCheck, it took 305 seconds (37.2 times longer). Note that this is a worst-case evaluation as the application is completely CPU-bound. (Also note that we discuss optimization techniques at the end of this section, one of which in early implementation improves performance overhead to 24 times slowdown.)

Short-lived: cfingerd When a program starts, each basic block is rewritten on demand to include TaintCheck's instrumentation. While basic block caching amortizes this penalty over a long execution time, it can be more significant for very short-lived processes. In order to evaluate this case, we timed how long cfingerd 1.4.2 takes to start and serve a finger request. cfingerd runs under inetd, which means it restarts for each request.

Without Valgrind, the request took an average of .0222 seconds. Using the Nullgrind skin, the request took 13 times as long. The Memcheck skin took 32 times as long, and TaintCheck took 36 times as long.

Common case: Apache For many network services, the latency that a user experiences is due mostly to network and/or disk I/O. For these types of services, we expect that the TaintCheck's performance penalty should not be as noticeable to the user. To evaluate this type of workload, we used the Apache 2.0.49 web server.

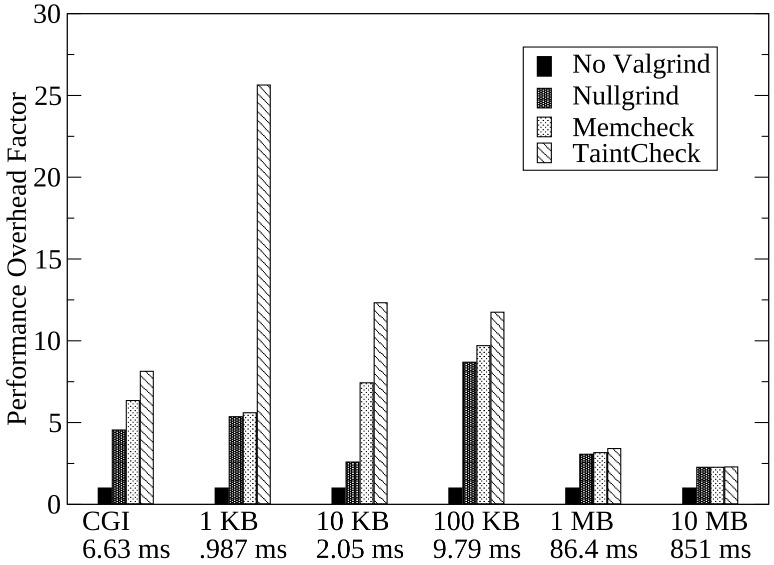


Figure 3: Performance overhead for Apache. Y-axis is the performance overhead factor: execution time divided by native execution time. Native execution times are listed below each experiment.

In these tests we requested different web pages from the server, and timed how long it took to connect, send the request, and receive the response. In order to prevent resource contention between the client process and the server process, the client was run from another machine connected to the server by a 100 Mbps switch. We requested a dynamically generated CGI shell script and static pages of sizes 1 KB to 10 MB. For each test, we requested the same page one hundred times, (thus allowing the document to stay in the server's cache) and used the median response time. Figure 3 shows the performance overhead for each type of request.

We expected the overhead for the shell script to be relatively large, since the web server must spawn a shell and execute the script each time it is requested. Thus, Valgrind must retranslate and reinstrument the code for the shell on each request. Despite this, the performance overhead was roughly on par with the results for static page requests. For static page requests we found that the performance overhead was relatively small. As we expected, the penalty for running under Valgrind grows less apparent as the size of the request grows. This is because the server becomes less processor-bound, and more I/O bound. Note that even in the worst case of a small, 1 KB page, TaintCheck only causes the response time to be approximately 25 ms instead of 1 ms on a local, high-speed network. This delay is unlikely to be very noticeable to a user, especially if the page were being loaded over a wide area network.

Improving performance Note that the current implementation is a research prototype and is not yet optimized for performance. There are several ways that we can improve the performance of TaintCheck. First, some performance overhead is due to the implementation of Valgrind. We used Valgrind because it is open source and relatively easy to use. However, as we showed in our evaluation, programs run several times slower under Valgrind even when no additional instrumentation is performed. DynamoRIO [3], offers much better performance than Valgrind, due to better caching and other optimization mechanisms. According to Kiriansky et. al. [27], DynamoRIO causes a bzip2 benchmark to run approximately 1.05 times slower than when run natively. Our tests show that bzip2 runs 3.1 times slower under Valgrind then when run natively. As we show next, TaintCheck runs significantly faster when implemented on DynamoRIO.

Second, when performing instrumentation, we could statically analyze each basic block to eliminate redundant tracking code. This optimization would significantly reduce the amount of instrumentation added, causing the instrumented program to run significantly faster. A preliminary implementation of this optimization allowed our bzip2 benchmark to run only 24 times slower than native speed, instead of 37 times slower as in our non-optimized implementation. We expect that further optimization could yield an even larger performance improvement.

Windows/DynamoRIO implementation Performance Analysis The Webstone 2.5 benchmark suite was used to benchmark the performance overhead of the Windows/DynamoRIO implementation of TaintCheck on Apache 1.3.23. The results are given in Table 2. The benchmark tests were run several times with increasing numbers of simultaneous clients. The TaintCheck enabled server was able to serve all client requests despite the increasing number of clients, but had noticeably lower throughput.

The Connection Rate entries show the number of connections that the server was able to handle per second. The Response Time is the average amount of time needed to transfer all the data requested by each client after the connection has already been established. The total number of megabits transfered to all clients by the server is the Throughput.

The benchmark indicates that TaintCheck caused the Apache server to

	Baseline Performance			**TaintCheck Enabled Performance**		
Clients	Connection Rate (/s)	Response Time (s)	Throughput (Mbits)	Connection Rate (/s)	Response Time (s)	Throughput (Mbits)
20	362.31	0.0550	57.36	57.77	0.3450	9.50
30	342.69	0.0870	53.74	60.31	0.4960	9.64
40	342.58	0.1160	54.43	59.74	0.6690	9.86
50	342.33	0.1460	54.62	60.51	0.8250	9.50
60	342.01	0.1740	54.07	60.80	0.9850	9.38
70	341.86	0.2040	53.60	60.72	1.1500	9.49
80	341.46	0.2330	53.30	61.01	1.3090	9.40
90	339.30	0.2640	53.96	60.52	1.4820	9.60
100	340.21	0.2930	53.49	60.95	1.6370	9.30

Table 2: Windows/DynamoRIO implementation performance analysis

run 5.7 times slower, since each metric is 5.7 times larger for a TaintCheck enabled server than for a vanilla Apache server.

5 Detection and analysis of new worm attacks

TaintCheck's properties make it a valuable tool for detecting new attacks. An early warning of a new worm or exploit can buy time to enable other countermeasures, such as generating and disseminating an attack signature to filter traffic, and eventually, patching vulnerable systems. In Section 6, we show how TaintCheck can also assist in creating a signature for a new attack so that it can be filtered out at network perimeters. In this section, we describe the usage scenario for TaintCheck. TaintCheck can be used at an individual site where it can either be used in conjunction with other detectors to reduce their false positives rate, or to provide additional attack information; or be used independently to check sampled requests. To enable faster detection of a new worm, TaintCheck can also be used in a distributed setting.

5.1 Individual usage

Individual sites can use TaintCheck to detect or prevent attacks on themselves.[2] Ideally, a site could use TaintCheck to protect its services all of the

[2] Note that if TaintCheck does not detect an exploit, it could be because the particular version of the server being used is not vulnerable. In order to be certain that a request

time. However, this is impractical in many cases because of TaintCheck's performance overhead. While a more optimized implementation of TaintCheck could run much faster than the current implementation, there will always be some performance penalty for dynamic taint analysis.

There are several ways that TaintCheck could be used to detect new attacks. One approach is to use it in conjunction with a faster detector in order to reduce its false positive rate and provide additional information about an attack. In particular, we show how TaintCheck can be combined with honeypots, or with operating system randomization [14, 25, 11]. TaintCheck can also be used by itself, *sampling* requests when it is unable to keep up with all incoming requests. This approach could detect new attacks that other detectors may miss.

TaintCheck-enabled honeypots A honeypot is a network resource with no legitimate use. Any activity on a honeypot is likely to be malicious, making them useful for detecting new attacks [29]. However, not everything sent to a honeypot is necessarily an exploit. Requests could be relatively innocuous probes or sent by mistake by legitimate users. An attacker who discovers a honeypot could also raise false alarms by sending innocuous requests to the honeypot. This is particularly a problem if requests sent to the honeypot are used to automatically generate attack signatures.

A honeypot could use TaintCheck to monitor all of its network services. This would allow it to verify whether requests that it receives are exploits before deciding what action to take, and provide additional information about detected attacks.

TaintCheck plus randomization Several techniques have been proposed to randomize parts of the operating system, such as the location of the stack [14], the location of the heap [14, 11], the system call interface [14], or even the instruction set [25]. These techniques make it difficult for an attacker's code to run correctly on an exploited system, typically causing a program to crash once it has been exploited, thus minimizing the damage caused. However, these techniques alone cannot prevent future attacks. The attacker is free to attack the vulnerable program again and again, usually causing the program to crash, and possibly even exploiting the program if he is lucky enough to guess the randomized values [43]. Further, it is not

does *not* contain an exploit, the system needs to check against every version of the server that it is interested in protecting. An individual site can protect itself by checking only the versions of the server that it actually uses.

possible to identify which request caused the program to crash, or whether that request was actually an attack.

We can identify which request contained an attack, if any, by using TaintCheck to analyze a log of recent requests after a program crashes. Once an exploit request has been identified, it is possible to take a number of actions, including generating a signature for the attack, or simply blocking future requests from the sender.

TaintCheck with sampling We can use TaintCheck independently on randomly *sampled* incoming requests. Depending how the sampling is done, TaintCheck can be used to detect or prevent new attacks with probability proportional to the sampling rate. This is particularly important when other detection mechanisms fail to detect such new attacks.

In order to *prevent* attacks, sampled requests can be redirected to a server that is protected by TaintCheck (possibly on the same machine as the normal server). This approach has two drawbacks. First, legitimate requests that are sent to the protected server are served somewhat more slowly. However, for I/O-bound services, this difference may not be noticeable as shown in Section 4. Second, an attacker may be able to detect that he is using the protected server by measuring the response time. In that case, he may be able to abort his request before the exploit takes place, later resending it in hope that it would go to the unprotected server. For that reason, it may be desirable to choose what requests to sample on a per user basis rather than a per request basis.

The other approach is to allow sampled requests to use the normal server, and replay them in parallel to the TaintCheck protected server. In this case, when an attack is detected the unprotected server may have already been compromised. However, the system could immediately quarantine the potentially compromised server, and notify administrators, thus minimizing any damage.

In either case, once a new attack has been detected by TaintCheck, it is possible to prevent further attacks by using TaintCheck to protect the normal server (with a 100% sampling rate) until the vulnerability can be fixed or an attack signature can be used to filter traffic. We discuss using TaintCheck to help generate a signature in Section 6.

TaintCheck with anomaly detection We can improve upon random sampling by using network anomaly detection techniques to identify requests that are likely to be attacks. We can then bias the selection of what requests to monitor towards requests that are identified as anomalous. Note that carefully crafted attacks may be able to evade anomaly detection techniques, so we should still select some fraction of requests at random to check with TaintCheck to guarantee that a new attack spreading across the Internet will eventually be detected by a TaintCheck monitor. Some requests that could be flagged as anomalous are requests that:

- Are sent from sources that have been observed performing port scans [26].
- Contain byte patterns that are suddenly very common, and are seen going between many source/destination pairs [47].
- Are part of an anomalous communication pattern (*e.g.*, an *outbound* port 80 (HTTP) connection from your web server).
- Have an unusual distribution of byte values (*e.g.*, many non-ASCII bytes in a text-based protocol) [54].
- Contain executable code [52].

Compared to using purely randomly sampling, this technique can increase the probability that an attack is detected. Compared to using network anomaly detection techniques by themselves, this technique greatly reduces the risk of false positives.

5.2 Distributed usage

Sites using TaintCheck could also cooperate for faster attack detection. Once one site has detected a new attack, the information about the attack can be used by anyone to defend against the attack. Ideally, a signature for an attack could be generated as soon as one site running TaintCheck detects the attack. This signature could then be used by everyone to efficiently filter out attack requests, thus preventing further exploits.

As a concrete example, suppose that a worm author develops a hit list of vulnerable servers and hard codes it into a worm [48]. Such a worm could spread even more quickly than fast scanning worms such as Blaster. The worm author could also throttle the spread of the worm, which may allow it to infect more machines before the worm was detected than if it spread as quickly as possible. Whether by brute force or by stealth, such a worm could infect a very large number of machines before it was noticed. However, if TaintCheck is deployed on d fraction of the vulnerable servers, each of which samples requests with probability s, we would expect to detect the worm by the time that $\frac{1}{ds}$ vulnerable servers are compromised. For example, if 10% of the vulnerable servers sample 10% of their incoming traffic with TaintCheck, the worm should be detected around the time that 100 servers have been compromised. If there are 1 million vulnerable hosts, this means the new attack can be detected after only 0.01% vulnerable servers are compromised. By automatically generating and distributing a signature for the worm, further compromises of other vulnerable hosts would be significantly reduced.

6 Automatic signature generation

Once a new exploit or worm is detected, it is desirable to generate a signature for it quickly, so that exploit requests may be filtered out, until the vulnerability can be patched. We first propose a new approach for automatic signature generation: using automatic semantic analysis of attack payloads. We describe the advantages of this approach, and describe how it could be implemented using TaintCheck. We then show several ways that TaintCheck can be used as a classifier in order to enhance automatic signature generation systems (both existing ones using content pattern extraction, and future ones using automatic semantic analysis).

6.1 Automatic semantic analysis based signature generation

Previous automatic signature generation techniques use content pattern extraction to generate signatures [26, 29, 46]. That is, they consider attack payloads as opaque byte sequences, and attempt to find patterns that are constant across attack payloads to use as signatures.

We propose a new approach for automatic signature generation: using *automatic semantic analysis* of attack payloads to identify which parts of the payloads are likely to be constant (*i.e.*, useful in a signature). Semantic analysis could potentially allow an accurate signature to be generated given fewer attack payloads than are necessary in systems that use only content pattern extraction. By requiring fewer attack payloads, semantic analysis could generate a signature at an earlier stage of a worm epidemic, thus minimizing damage caused by a new worm.

TaintCheck could be used to perform automatic semantic analysis of attack payloads, because it monitors how each byte of each attack payload is used by the vulnerable program at the processor-instruction level. As a first step, we have implemented a feature that allows TaintCheck to identify the value used to overwrite a function pointer or return address. We also

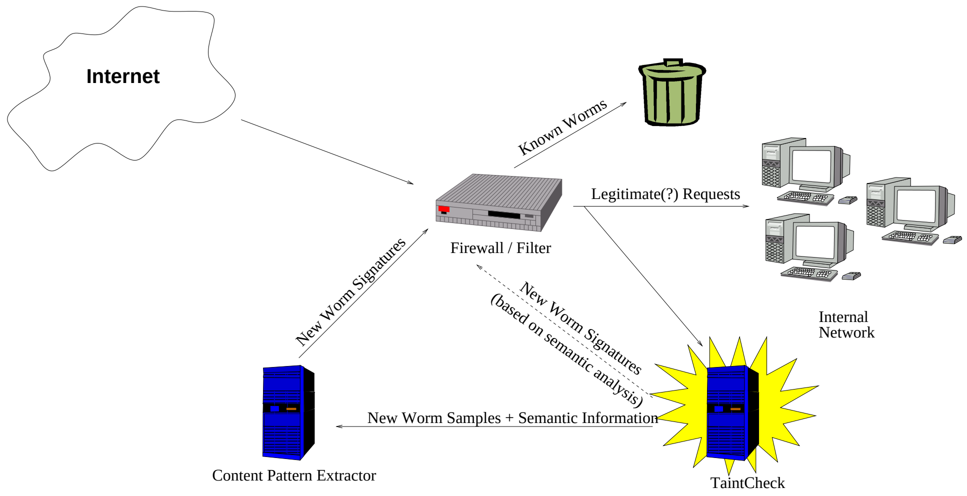

Internet
Known Worms
Legitimate(?) Requests
Firewall / Filter
Internal Network
New Worm Signatures
New Worm Signatures (based on semantic analysis)
New Worm Samples + Semantic Information
Content Pattern Extractor
TaintCheck

Figure 4: Using TaintCheck to detect new attacks and perform automatic semantic analysis.

describe several other promising directions for using TaintCheck to perform automatic semantic analysis.

Figure 4 illustrates how an automatic signature generation system could use TaintCheck to detect new attacks, and to perform semantic analysis of attack payloads. In some cases, the semantic information could be used directly as a signature. Otherwise, it can be used to give hints to a content pattern extractor, possibly allowing it to generate an accurate signature with fewer payloads than it would require otherwise.

Obtaining overwrite values In Section 4 we show that TaintCheck can identify the value used to overwrite a return address or a function pointer. For most code-injection exploits, this value needs to point to a fixed area for the exploit to work correctly; *i.e.*, usually at least the three most significant bytes of this value must remain constant. For many exploits, this value must occur literally in the attack payload. In other exploits, the server may perform a decoding step (such as URL decoding) in order to obtain the actual pointer. TaintCheck can distinguish between these two cases, and trace back to determine which bytes of the encoded payload correspond to the decoded address. Hence, TaintCheck is often able to produce a signature automatically based on the three most significant bytes of a value used to overwrite a jump target such as a return address or function pointer. Similarly, for existing-code attacks, there are typically only a few places that are useful for the attack to jump to (*e.g.*, a few global library entry points). Thus, similar techniques will work for identifying signatures for existing-code attacks as well.

While a three-byte signature may seem short, it could be specific enough for protocols with an uneven distribution of byte sequences (*e.g.*, text-based protocols such as HTTP). In our analysis of a week-long trace of incoming and outgoing HTTP requests containing 59250 connections from the Intel Research Pittsburgh Lab, 99.97% of all three-byte sequences occurred in less than .01% of all connections, and 91.8% of all three-byte sequences never occurred at all. To further test this idea, we used TaintCheck to identify the return address used in the ATPhttpd exploit discussed in Section 4.2.2. We found that the three most significant bytes of this value occurred in only one request in the week-long trace. The request was a POST request that was used to upload a binary file. This corresponds to a false positive rate of .0017%. Hence, the three most significant bytes of the value used to overwrite a return address or function pointer, which can be identified by TaintCheck, are often distinctive enough to be used as a reasonably accurate signature by themselves, at least until a more descriptive signature can be found or the vulnerability can be repaired. When it is not specific enough to be used by itself, it can still be used as part of a more specific signature. Note that our analysis is also consistent with the findings in [35], which offers a more extensive analysis of the usage of return addresses as signatures. In experiments using 19 different real exploits and traces, [35] demonstrates that using a range of return addresses (approximately the three most significant bytes) as a signature can filter out nearly 100% worm attack packets while only dropping 0.01% of legitimate packets.

Note that attacks exploiting a format string vulnerability may not have a constant return address that we can leverage for a signature, because a format string vulnerability often enables the attacker to overwrite any location in memory with any value. However, in order to perform the overwrite, the format string supplied by the attacker often needs to contain certain format modifiers such as %n. When TaintCheck detects a format string attack, it can determine whether the format string is directly copied from the attack packet; if so, then we could use such format modifiers and use this as part of the attack signature.

Potential techniques for further semantic analysis In future work, we plan to investigate more advanced techniques of semantic analysis to assist automatic signature generation.

One possibility is for TaintCheck to keep track of whether each byte of the request is used in any significant way, and how it is used. This could be helpful for identifying *filler* bytes, which could be (but don't have to be) completely different in each instance. These bytes could be completely random, or the attacker could use these bytes to form *red herring* patterns, fooling the system into generating useless or harmful signatures. If any byte is not used to affect a branch, used to perform a calculation, used in a system call, or executed by the worm, it is likely a filler byte. While an attacker may attempt to perform meaningless operations on the filler bytes in order to fool such a technique, it may be possible to extend the Exploit Analyzer with dynamic slicing techniques [7, 28] to identify which operations are "meaningful" and which are not. Additionally, any bytes used only *after* the exploit has taken place may not be essential to exploit the vulnerability. Hence, they could be different in other instances of a polymorphic worm (one that uses encryption and code obfuscation so that no two instances are identical [51]), or in different exploits for the same vulnerability. Using such bytes in a signature may make the signature more specific than it needs to be, leading to future false negatives. Conversely, bytes that *are* used by the program before it is exploited may be necessary to make the program follow the control path leading to the exploit.

Another technique that could be used to determine which parts of the request are irrelevant to the exploit is to flip bits in the attack packet and see whether the exploit still succeeds. If the attack can still succeed after a certain bit is flipped, then we will know that that the value of that bit will not affect the success of the exploit and hence may not be invariant for new attacks on the same vulnerability.

6.2 Classifier and signature verifier

In addition to automatic semantic analysis based signature generation, TaintCheck has direct applications to other aspects of automatic signature generation. TaintCheck can be used as a classifier in order to enhance automatic signature generation systems (both existing ones using content pattern extraction, and future ones using automatic semantic analysis). As we have shown in Section 5, TaintCheck can be used to detect new attacks so that signatures can be generated for them. TaintCheck could also be used to verify the quality of newly generated signatures, by determining whether requests matching a newly generated signature actually contain an attack.

Classifying attack payloads Previous automatic signature generation systems such as Earlybird [46], Honeycomb [29], and Autograph [26] use coarse grained attack detectors to find samples of new attacks. Each of these techniques can potentially identify innocuous requests as malicious requests, either by accident, or as the result of an attacker "feeding" the system innocuous requests (*e.g.*, by sending them to a known honeypot). When this happens, a signature could be generated that matches legitimate requests, causing a denial of service attack if that signature is used to filter traffic. Moreover, these systems need to wait until multiple (potentially many) payloads are captured before generating a signature, in order to decrease the probability of a signature being generated for legitimate requests that were incorrectly identified as worms. Thus, they suffer from a tradeoff between false positive rate and speed of signature generation.

As we have shown in Section 5, TaintCheck can be used either by itself or in conjunction with other classifiers to help accurately detect new attacks. The system can be nearly certain that a request is malicious if TaintCheck determines that an exploit occurred, since TaintCheck offers a very low incidence of false positives. Hence, there is no need to wait for other similar requests to be classified as malicious to be confident that the request is actually malicious, as done in previous signature-generation systems.

Another technique that we have implemented in TaintCheck is an option to allow a worm to attempt to spread in a confined environment after it has exploited the protected server, while redirecting all outgoing connections to a logging process. In this way, any number of samples can be generated from just one worm sample. These additional samples can be used to help identify which parts of the worm are invariant across different instances of the worm. These invariant portions can then be used to generate a signature.

In future work, we plan to investigate more advanced techniques for TaintCheck to further assist in automatic signature generation as a classifier. For example, TaintCheck can not only detect exploit attacks, but also distinguish between different vulnerabilities and different exploits. Thus, TaintCheck can not only be a one-bit classifier, *i.e.*, whether a payload contains an exploit attack or not, but also be a more sophisticated classifier, *i.e.*, classify different payloads into different groups according to the vulnerability and the exploit. Pattern extraction methods can then be used in each group, and thus, generate more accurate signatures.

Signature and alarm verification TaintCheck can also be used to verify signatures and alarms. In a single-user setting, this can be used to verify the quality of a newly generated signature. This can be done by sending incoming requests that match the signature to a TaintCheck monitored version of the server, instead of just dropping them. False positives can then be measured as the fraction of incoming requests that match the signature that actually turn out to be attacks. In a distributed system where attack signatures and alarms to new attacks are disseminated, attackers or incompetent participants may inject bogus or low quality signatures and alarms that will cause denial-of-service attacks on legitimate traffic. Thus, the receiver needs to validate the signatures and the alarms received to ensure that they are valid and will not cause denial-of-service attacks. TaintCheck can be used as the verifier to check that the remotely generated signatures and alarms are valid. If samples of the attack are distributed along with the signature, TaintCheck can verify that the samples actually contain attacks, and hence that there actually is a threat. It can also be used to measure the false positive rate of the signature, in the same way as in the single-user case.

7 Related work

Program Shepherding. Program Shepherding [27] is the closest related work to TaintCheck. Program Shepherding is a runtime monitoring system that keeps track of whether code has been modified since it was loaded from disk, and checks each control transfer to ensure that the destination is to a basic block that has not been modified. Thus, Program Shepherding can prevent code injection attacks. It also prevents some existing code attacks by ensuring that control transfers to a library can only go to exported entry points, and that `return` addresses point to just after a `call` instruction. However, these techniques do not prevent many existing-code attacks (*e.g.*, overwrite a function pointer to the `exec` library call). In contrast, TaintCheck can prevent these existing-code attacks. Moreover, TaintCheck, via dynamic taint analysis, provides detailed information how the vulnerability is exploited. Program Shepherding does not.

Other runtime detection mechanisms Many approaches have been proposed to detect when certain vulnerabilities are exploited by attacks. Most of these previous mechanisms require source code or special recompilation of the program, such as StackGuard [19], PointGuard [18], full-bounds check [24, 42], LibsafePlus [8], FormatGuard [17], and CCured [32]. Many of them require recompiling the libraries [24, 42], modifying the original source code, or are not compatible with some programs [32, 18]. These issues hinder the deployment and the applicability of these methods in attack defense for commodity software, since source code is often unavailable.

Several other approaches for runtime attack detection have been proposed that do not require source code or specially compiled binary programs, such as LibSafe [9], LibFormat [41], Program Shepherding [27], and the Nethercote-Fitzhardinge bounds check [33]. However, they fail to detect many types of attacks. For example, LibSafe only catches buffer overflows related to certain string-handling functions, LibFormat only detects certain format modifiers in format strings and thus can have high false positives and false negatives, and the Nethercote-Fitzhardinge bounds check has significant false positives and false negatives. In contrast, TaintCheck detects a wider range of attacks and incurs fewer false positives.

Other approaches have been proposed for more coarse-grained attack detection, including system call interposition (e.g. Systrace [38], GSWTK [22], Tron [10], Janus [23], and MAPbox [6]). These approaches detect attacks by detecting anomalous system call sequences in a program. They do not give detailed information about the vulnerability and how it is exploited, and require building models for each protected program.

Other taint-based approaches Static taint analysis has been used to find bugs in C programs [21, 44, 55] or to identify potentially sensitive data in Scrash [12]. Perl [5] does runtime taint checking to see whether data from untrusted sources are used in security-sensitive ways such as as an argument to a system call.

Chow *et. al.* independently and concurrently proposed to use whole-system simulation with tainting analysis to analyze how sensitive data are handled in large programs such as Apache and Emacs [15]. The tainting propagation in TaintCheck is similar to the one in [15]. However, their work focuses on analyzing the lifetime of sensitive data such as passwords, where our work concerns attack detection and defense.

Minos independently and concurrently proposed hardware extension to perform Biba-like data integrity check of control flow to detect attacks at runtime [20]. Their work uses hardware and OS modifications to perform Biba integrity checks at the whole-system level. In contrast, TaintCheck requires no hardware or OS modifications, and can be very flexible and fine-grained to detect attacks that were not addressed in Minos such as format string vulnerabilities[3] and attacks that overwrite security-sensitive

[3] Minos can detect some forms of format string vulnerabilities if they alter the control flow, however, our work can detect format string vulnerabilities even when they do not

variables such as system call arguments. TaintCheck is also able to perform more detailed analysis of detected attacks, which can be used for automatic signature generation.

Suh *et. al.* independently and concurrently proposed a hardware extension to detect attacks by detecting when data from untrusted inputs is used as instructions or jump target addresses [49].

Vigilante [16] is a position paper that proposes using dynamic taint analysis in an automated host-based worm containment system.

Other signature generation approaches and defense systems Several automatic signature generation methods have recently been proposed, including Earlybird [46], Honeycomb [29], and Autograph [26]. Earlybird [46] monitors traffic and generates signatures which consist of the Rabin fingerprints of the most commonly occurring 39 byte substrings in the monitored traffic. Honeycomb [29] classifies traffic that is sent to a honeypot as malicious, and generates signatures based on the longest common substrings found in traffic sent to the honeypot. Autograph [26] uses distributed monitoring points to determine what hosts are performing port scans. All traffic from hosts that have been seen performing port scans is classified as malicious. Autograph then uses a file chunking technique to split requests into blocks, and generates a signature consisting of the most common blocks seen in malicious traffic.

As we showed in Section 6, TaintCheck can be used as a classifier to reduce the false positive and/or false negative rates of the classifiers used in these systems. TaintCheck can also provide semantic analysis of attack payloads, which can be used to generate signatures with fewer samples than by using content-analysis alone. Finally, TaintCheck can also be used to verify the signatures and alarms produced by such systems.

Shield [53] presents an alternative approach to content-based filtering. Shield uses the characteristics of a vulnerability to manually generate a signature for any exploit of that vulnerability *before* an exploit is seen in the wild. This is a promising approach, but it does not help in the case that the worm utilizes a vulnerability that was previously unknown, or only recently became known.

Sidiroglou and Keromytis propose a worm vaccine architecture to automatically generate patches for vulnerabilities [45].

alter control flow.

8 Conclusion

In order to combat the rapid spread of a new worm before it can compromise a large number of machines, it is necessary to have automatic attack detection and defense mechanisms. In this paper we have proposed *dynamic taint analysis* and shown how it can be used to detect and analyze most types of software exploits, without requiring source code or special compilation of a program, thus allowing it to easily be used on commodity software. It reliably detects many attacks, and we have found no false positives in our tests. In addition, because it monitors the execution of a program at a fine-grained level, TaintCheck can be used to provide additional information about the attack. It is currently able to identify the input that caused the exploit, show how the malicious input led to the exploit at a processor-instruction level, and identify the value used to overwrite the protected data (*e.g.* the return address).

Furthermore, we have shown that TaintCheck is particularly useful in an automatic signature generation system—it can be used to enable semantic analysis based signature generation, enhance content pattern extraction based signature generation, and verify the quality of generated signatures.

9 Acknowledgments

We would like to thank David Brumley, Mike Burrows, Jedediah Crandall, Debin Gao, Brad Karp, Angelos Keromytis, Nicholas Nethercote, Jonathon Shapiro, Peter Szor, Helen Wang, Felix Wu, Avi Yaar, and Lidong Zhou for providing feedback and assistance on this project. We also thank the anonymous reviewers for their insightful feedback.

References

[1] Cert advisory ca-2002-27 apache/mod_ssl worm. `http://www.cert.org/advisories/CA-2002-27.html`.

[2] Cert advisory ca-2003-04 ms-sql server worm. `http://www.cert.org/advisories/CA-2003-04.html`.

[3] Dynamorio. `http://www.cag.lcs.mit.edu/dynamorio/`.

[4] ISC innd 2.x remote buffer overflow vulnerability. `http://securityfocus.com/bid/1316`.

[5] Perl security manual page. `http://www.perldoc.com`.

[6] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. In *the Proceedings 9th USENIX Security Symposium*, 2000.

[7] Hiralal Agrawal and Joseph Horgan. Dynamic program slicing. In *Proc. SIGPLAN*, 1990.

[8] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, August 2004.

[9] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent runtime defense against stack smashing attacks. In *USENIX Annual Technical Conference 2000*, 2000.

[10] Andrew Berman, Virgil Bourassa, and Erik Selberg. Tron: Process-specific file protection for the unix operating system. In *the Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, 1995.

[11] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.

[12] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A system for generating security crash information. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.

[13] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *International Conference on Compiler Construction*, April 2003.

[14] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[15] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, August 2004.

[16] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain internet worms? In *HotNets 2004*, 2004.

[17] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[18] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, 2003.

[19] Crispin Cowan, Calton Pu, Dave Maier, Jonathon Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[20] Jedidiah R. Crandall and Frederic T. Chong. Minos: Architectural support for software security through control data integrity. In *To appear in International Symposium on Microarchitecture*, December 2004.

[21] Jeffrey Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[22] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *the Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[23] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *the Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, USA, 1996.

[24] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging*, 1995.

[25] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and Communication Security*, 2003.

[26] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[27] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[28] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. In *J. Systems and Software*, volume 13, 1990.

[29] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[30] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. In *IEEE Security and Privacy*, volume 1, 2003.

[31] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *2003 IEEE Infocom Conference*, 2003.

[32] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*, 2002.

[33] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, January 2004. (Proceedings not formally published.).

[34] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

[35] Archana Pasupulati, Jason Coit, Karl Levitt, and Felix Wu. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *IEEE/IFIP Network Operation and Management Symposium*, May 2004.

[36] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24), December 1999.

[37] The Snort Project. Snort, the open-source network intrusion detection system. `http://www.snort.org/`.

[38] Niels Provos. Improving host security with system call policies. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.

[39] r code. ATPhttpd exploit. `http://www.cotse.com/mailing-lists/todays/att-0003/01-atphttp0x06.c`.

[40] Yann Ramin. ATPhttpd. `http://www.redshift.com/~yramin/atp/atphttpd/`.

[41] Tim J Robbins. libformat. `http://www.securityfocus.com/tools/1818`, 2001.

[42] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[43] Hovav Shacham, Mattew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address space randomization. In *ACM Computer and Communication Security Symposium*, 2004.

[44] Umesh Shankar, Kunal Talwar, Jeffrey Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[45] Stelios Sidiroglou and Angelos D. Keromytis. A network worm vaccine architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.

[46] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The EarlyBird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, August 2003.

[47] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[48] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. In *11th USENIX Security Symposium*, 2002.

[49] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, October 2004.

[50] Cisco Systems. Network-based application recognition. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newf%t/122t/122t8/dtnbarad.htm`.

[51] Peter Szor. Hunting for metamorphic. In *Virus Bulletin Conference*, 2001.

[52] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advance in Intrusion Detection (RAID)*, October 2002.

[53] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, August 2004.

[54] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advance in Intrusion Detection (RAID)*, Sep 2004.

[55] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQual for static analysis of authorization hook placement. In *the Proceedings of the 11th USENIX Security Symposium*, 2002.