# Pointless Tainting?
# Evaluating the Practicality of Pointer Tainting

Asia Slowinska
Vrije Universiteit Amsterdam
asia@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam and NICTA *
herbertb@cs.vu.nl

## Abstract

This paper evaluates pointer tainting, an incarnation of Dynamic Information Flow Tracking (DIFT), which has recently become an important technique in system security. Pointer tainting has been used for two main purposes: detection of privacy-breaching malware (e.g., trojan keyloggers obtaining the characters typed by a user), and detection of memory corruption attacks against non-control data (e.g., a buffer overflow that modifies a user's privilege level). In both of these cases the attacker does not modify control data such as stored branch targets, so the control flow of the target program does not change. Phrased differently, in terms of instructions executed, the program behaves 'normally'. As a result, these attacks are exceedingly difficult to detect. Pointer tainting is considered one of the only methods for detecting them in unmodified binaries. Unfortunately, almost all of the incarnations of pointer tainting are flawed. In particular, we demonstrate that the application of pointer tainting to the detection of keyloggers and other privacy-breaching malware is problematic. We also discuss whether pointer tainting is able to reliably detect memory corruption attacks against non-control data. We found that pointer tainting generates itself the conditions for false positives. We analyse the problems in detail and investigate various ways to improve the technique. Most have serious drawbacks in that they are either impractical (and incur many false positives still), and/or cripple the technique's ability to detect attacks. In conclusion, we argue that depending on architecture and operating system, pointer tainting may have some value in detecting memory corruption attacks (albeit with false negatives and not on the popular x86 architecture), but it is fundamentally not suitable for automated detecting of privacy-breaching malware such as keyloggers.

***Categories and Subject Descriptors*** D.4.6 [*Security and Protection*]: Invasive software

***General Terms*** Security, Experimentation

***Keywords*** dynamic taint analysis, pointer tainting

## 1. Introduction

Exploits and trojans allow attackers to compromise machines in various ways. One way to exploit a machine is to use techniques like buffer overflows or format string attacks to divert the flow of execution to code injected by the attacker. Alternatively, the same exploit techniques may attack non-control data [Chen 2005b]; for instance a buffer overflow that modifies a value in memory that represents a user's identity, a user's privilege level, or a server configuration string. Non-control data attacks are even more difficult to detect than attacks that divert the control flow. After all, the program does not execute any foreign code, does not jump to unusual places, and does not exhibit strange system call patterns or any other tell-tale signs that indicate that something might be wrong.

While protection for some of these attacks may be provided if we write software in type-safe languages [Jim 2002], compile with specific compiler extensions [Castro 2006, Akritidis 2008], or verify with formal methods [Elphinstone 2007], much of the system software in current use is written in C or C++ and often the source of the software is not available, and recompilation is not possible.

Worse, even with the most sophisticated languages, it is difficult to stop users from installing trojans. Often trojans masquerade as useful programs, like pirated copies of popular applications, games, or 'security'-tools, with keylogging, privacy theft and other malicious activities as hidden features. No exploit is needed to compromise the system at all. Once inside, the malware may be used to join a spam botnet, damage the system, attack other sites, or stealthily spy on a user. Again, stealthy spies are harder to detect than 'loud'

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.



*EuroSys '09,* 1–3, April 2009, Nuremberg, Germany.

programs that damage systems, or engage in significant network activity. The trojan spyware, installed by the user, may use legitimate APIs to obtain and store the characters that are typed in by the users (or data in files, buffers, or on the network). From a system's perspective, the malware is not doing anything 'wrong'.

In light of the above, we distinguish between attacks that divert the control flow of a program and those that do not. *Control diversion* typically means that a pointer in a process is manipulated by an attacker so that when it is dereferenced, the program starts executing instructions different from the ones it would normally execute at that point. *Non control diverting* attacks, on the other hand, include memory corruption attacks against non-control data and privacy breaching malware like keyloggers and sniffers. Memory corruption attacks against non-control data manipulate data values that are not directly related to the flow of control; for instance, a value that represents a user's privilege level, or the length in bytes of a reply buffer. The attack itself does not lead to unusual code execution. Rather, it leads to elevated privileges, or unusual replies. The same is true for privacy breaching malware like sniffers and trojan keyloggers.

***Pointer tainting as advertised is attractive.*** It is precisely these difficult to detect, stealthy non-control-diverting attacks that are the focus of pointer tainting [Chen 2005a]. At the same time, the technique works against control-diverting attacks also. We will discuss pointer tainting in more detail in later sections. For now, it suffices to define it as a form of dynamic information flow tracking (DIFT) [Suh 2004] which marks the origin of data by way of a taint bit in a shadow memory that is inaccessible to software. By tracking the propagation of tainted data through the system (e.g., when tainted data is copied, but also when tainted pointers are dereferenced), we see whether any value derived from data from a tainted origin ends up in places where it should never be stored. For instance, we shall see that some projects use it to track the propagation of keystroke data to ensure that untrusted and unauthorised programs do not receive it [Yin 2007]. By implementing pointer tainting in hardware [Dalton 2007], the overhead is minimal.

Pointer tainting is very popular because (a) it can be applied to unmodified software without recompilation, and (b) according to its advocates, it incurs hardly (if any) false positives, and (c) it is assumed to be one of the only (if not *the* only) reliable techniques capable of detecting both control-diverting and non-control-diverting attacks without requiring recompilation. Pointer tainting has become a unique and extremely valuable detection method especially due to its presumed ability to detect non-control-diverting attacks. As mentioned earlier, non-control-diverting attacks are more worrying than attacks that divert the control flow, because they are harder to detect. Common protection mechanisms like address space randomisation and stackguard [Bhatkar 2005, Cowan 1998] present in several modern operating systems are ineffective against this type of attack. The same is true for almost all forms of system call monitoring [Provos 2003, Giffin 2004]. As a result, some trojan keyloggers have been active for years (often undetected). In one particularly worrying case, a keylogger harvested over 500,000 login details for online banking and other accounts [Raywood 2008]. At the same time, the consequences of a successful non-control-diverting attack may be as severe as with a control-diverting attack. For instance, passwords obtained by a keylogger often give attackers full control of the machines. The same is true for buffer overflows that modify a user's privilege level.

***However, pointer tainting is not working as advertised.*** Inspired by a string of publications about pointer tainting in top venues [Chen 2005a;b, Yin 2007, Egele 2007, Dalton 2007, Yin 2008, Venkataramani 2008, Dalton 2008], several of which claim zero false positives, we tried to build a keylogger detector by means of pointer tainting. However, what we found is that for privacy-breaching malware detection, the method is flawed. It incurs both false positives and negatives. The false positives appear particularly hard to avoid. There is no easy fix. Further, we found that almost all existing applications of pointer tainting to detection of memory corruption attacks are also problematic, and none of them are suitable for the popular x86 architecture and Windows operating system.

In this paper, we analyse the fundamental limitations of the method when applied to detection of privacy-breaching malware, as well as the practical limitations in current applications to memory corruption detection. Often, we will see that the reason is that 'fixing the method is breaking it': simple solutions to overcome the symptoms render the technique vulnerable to false positives or false negatives.

Others have discussed minor issues with projects that use pointer tainting [Dalton 2006], and most of these have been addressed in later work [Dalton 2008]. To the best of our knowledge, nobody has investigated the technique in detail, nobody has shown that it does not work against keyloggers, and we are the first to report the complicated problems with the technique that are hard to overcome. We are also the first to evaluate the implications experimentally.

In summary, the contributions of this paper are:

1. an in-depth analysis of the problems of pointer tainting on real systems which shows that it does not work against malware spying on users' behaviour, and is problematic in other forms also;
2. an analysis and evaluation of all known fixes to the problems that shows that they all have serious shortcomings.

We emphasise that this paper is not meant as an attack on existing publications. In our opinion, previous papers underestimated the method's problems. We hope that our work wlll help others avoid making the mistakes we made

```
struct req {
 char reqbuf[64];
 void (*handler)(char *);
};

void do_req(int fd,
       struct req *r)
{
 // now the overflow:
 read (fd,r->reqbuf,128);
 r->handler (r->reqbuf);
}
```

**(a) control data attack**

```
void serve (int fd)
{
 char *name = globMyHost;
 char cl_name[64];
 char svr_reply[1024];

 // now the overflow:
 read(fd,cl_name,128);
 sprintf(svr_reply,
        "hello %s, I am %s",
        cl_name, name);
 svr_send(fd,svr_reply,1024);
}
```

**(b) non-control data attack**

**Figure 1.** Trivial overflow examples

when we worked on our ill-fated keylogger detector, and perhaps develop improved detection techniques.

## 2. Threat Model

Before we can evaluate pointer tainting, we revisit in more detail the nature of the attacks that we introduced informally in the previous section. Recall that we said that we would distinguish between two types of attack: (1) control-diverting, and (2) non-control-diverting. Moreover, within the latter category we will distinguish between (2a) memory corruption attacks against non-control data, and (2b) privacy breaching malware, such as keyloggers and sniffers. We now define what they are.

Attackers often compromise computer systems by exploiting security vulnerabilities resulting from low-level memory errors such as buffer overflows, dangling pointers, and double frees. **Control-diverting attacks** exploit buffer overflows or other vulnerabilities to manipulate a value in memory data that is subsequently loaded in the processor's program counter (e.g., return addresses or function pointers) with the aim of executing either code that was injected by the attackers, or a particular library function. An example of an attack against control data is shown in Figure 1(a): a stylised server reads a request in a struct's buffer field and subsequently calls the corresponding handler. By overflowing `reqbuf`, an attacker may change the handler's function pointer and thus cause the program to call a function at a different address.

Non-control-diverting memory corruption attacks exploit similar vulnerabilities to modify security-critical data in ways that do not result in a different control flow. For instance, a buffer overflow on a server may overwrite the pointer to (part of) the reply message. As a result, an attacker controls the memory area used for the reply, possibly causing the server to leak confidential information. This example is shown in stylised form in Figure 1(b), which shows a trivial greeting server. To keep it simple, we use an overflow on the stack and assume that the program is compiled without stack protection. The server stores a pointer to its own name (which is defined as global string) in the variable `name` and then reads the name of the client from a socket. These two names are combined in a greeting message which is echoed to the client. If a malicious client overflows the `cl_name` buffer, it may overwrite the server's `name` pointer, which means that the reply string is composed of the client's string and a memory region specified by the attacker. The result is that information leaks out of the system.

As the instructions that are executed are exactly the same as for a non-malicious attack, this is an example of a non-control-diverting attack. For brevity, we will refer to them as **non-control data attacks** in the remainder of this paper. The other manifestation of the non-control-diverting class of attacks that we will look at concerns privacy breaching malware like keyloggers, spyware, and network sniffers.

In many ways, the nature of **privacy breaching malware** is completely different from the two types of attack discussed above, as it is not about intrusion itself. The malware may be installed by way of exploits, or as part of trojans downloaded by the users, or any other means. Once installed, it often uses legitimate means (e.g., existing APIs) to achieve illegitimate goals (theft of security sensitive information). As a result, techniques that detect intrusions are powerless. For instance, a keylogger in Windows often uses well-known OS APIs like `GetAsyncKeyState()`, or `GetForegroundWindow()`, to poll the state of the keyboard or to subscribe to keyboard events. In practice, a lot of spyware is implemented as a browser helper object (BHO) library that extends Internet Explorer. Since it runs in the same address space as the browser, it has full control over the browser's functionality. `Zango` [ProcessLibrary.com, Egele 2007], for instance, copies visited URLs to a shared memory section which is later read by a spyware helper process.

Again, the execution of the program that is spied upon does not change, and so we also classify these attacks as non-control-diverting. For convenience, this paper often uses keyloggers as an example, but we stress that the analysis holds for all types of privacy breaching malware. We do not care whether the malware is installed by the user, or by means of a prior exploit; nor do we care about the method that malware employs to access sensitive data. Our main interest is whether we are able to detect them as malware, and say that they access data that was not intended for them.

Since pointer tainting was originally designed to deal with non-control-diverting attacks (non-control data exploits and privacy breaching malware), we will concentrate on them rather than control-diverting attacks. We have already argued that these are the 'hard cases' anyway.

## 3. Pointer tainting

Pointer tainting is a variant of dynamic taint analysis, a technique for detecting various attacks. We show that it was originally proposed because taint analysis in its basic form cannot handle non-control-diverting attacks.

### 3.1 Basic tainting

One of the most reliable methods for detecting control diversions is known as dynamic taint analysis. The technique marks (in an emulator or in hardware) all data that comes from a suspect source, like the network, with a taint tag. The tag is kept in separate (shadow) memory, inaccessible to the software. Taint is propagated through the system to all data derived from tainted values. Specifically, when tainted values are used as source operands in ALU operations, the destinations are also tainted; if they are copied, the destinations are also tainted, etc. Other instructions explicitly 'clean' the tag. An example is '`xor eax,eax`' on x86 which sets the `eax` register to zero and cleans the tag. An alert is raised when a tainted value is used to affect a program's flow of control (e.g., when it is used as a jump target or as an instruction). We summarise the rules for taint propagation:

1. all data from suspect sources is tainted;
2. when tainted data is copied, or used in arithmetical calculations, the taint propagates to the destination;
3. taint is removed when all traces of the tainted data are removed (e.g., when the bytes are loaded with a constant) and a few other operations.

Basic taint analysis has been successfully applied in numerous systems [Crandall 2004, Newsome 2005, Costa 2005, Ho 2006, Portokalidis 2006, Slowinska 2007, Portokalidis 2008]. The drawback is that it protects against control-diverting attacks, but not against non-control-diverting attacks as shown presently.

***Memory corruption and the (in)effectiveness of basic tainting.*** For exploits, the root cause of almost all control-diverting *and* non-control data attacks is the dereference of attacker-manipulated pointers. For instance, a stack smashing attack overflows a buffer on the stack to change the function's return address. Similarly, heap corruption attacks typically use buffer overflows or double frees to change the forward and backward links in the doubly linked free list. Alternatively, buffer overflows may overwrite function pointers on heap or stack directly. In a format string attack, a member of the `printf()` family is given a specially crafted format string to trick it into using an attacker-provided value on the stack as a pointer to an address where a value will be stored.

The nature of these attacks vary, but they all rely on dereferencing a pointer provided by the attacker via memory corruption. Basic taint analysis raises alerts only for dereferences due to jumps, branches, and function calls/returns. A modification of a value representing a user's privilege level in a non-control data attack would go unnoticed.

***Privacy-breaching and the ineffectiveness of basic tainting.*** One may want to employ dynamic taint analysis to detect whether a 'possibly malicious' program is spying on users' behaviour. A basic approach could work by marking the keystrokes typed by the user as tainted, and monitoring the taint propagation in order to inspect whether the software in question accesses tainted sensitive data.

However basic taint analysis is weak in the face of translation tables that are frequently used for keystrokes. Assuming variable $x$ is tainted, basic taint analysis will not taint $y$ on an assignment such as $y = a[x]$, even though it is completely dependent on $x$. As a practical consequence, data from the keyboard loses its taint almost immediately, because the scan codes are translated via translation tables. The same is true for ASCII/UNICODE conversion, and translation tables in C library functions like `atoi()`, `to_upper()`, `to_lower()`, `strtol()`, and `sprintf()`.

As a corollary, basic taint analysis is powerless in the face of privacy-breaching malware. As data loses its taint early on, it is impossible to track if it ends up in the wrong places.

### 3.2 Pointer tainting

Pointer tainting is explicitly designed to handle non-control-diverting attacks. Because of the two different application domains, pointer tainting comes in two guises, which we will term *limited pointer tainting* (for detecting non-control data attacks) and *full pointer tainting* (for detecting privacy breaches). Both have shortcomings. To clarify the problems, we first explain the two variants in detail. For now, we just describe the basic ideas. We will see later that they both need to be curtailed to reduce the number of false positives.

***Limited pointer tainting (LPT): alerts on dereferences.*** Systems that aim at detecting non-control data attacks apply a limited form of pointer tainting [Chen 2005a, Dalton 2008]. Defining a tainted pointer as an address that is generated using tainted data, taint analysis is extended by raising an alert when a tainted pointer is dereferenced. So:

4a. if $p$ is tainted, raise an alert on any dereference of $p$.

Doing so catches several of the memory corruption exploits discussed above, but cannot be realistically applied in the general case. For instance, any pointer into an array that is calculated by way of a tainted index would lead to an alert when it is dereferenced, causing false positives. Again, this is common in translation tables. For this reason, LPT implementations in practice prescribe that the taint of an index used for a safe table lookup is cleaned. In Section 6.2 we evaluate various such cleaning techniques. As a consequence, however, LPT cannot be used for tracking keystrokes. As soon as the tainted keystroke scan-code is converted by a translation table, the taint is dropped and we lose track of the sensitive data.

***Full pointer tainting (FPT): propagation on dereferences.*** Full pointer tainting extends basic taint analysis by propagating taint much more aggressively. Rather than raising an alert, pointer tainting simply propagates taint when a tainted pointer is dereferenced. So:

4b. if $p$ is tainted, any dereference of $p$ taints the destination.

FPT looks ideal for privacy-breaching malware detection; table conversion preserves the original taint, allowing us to track sensitive data as it journeys through the system. Panorama [Yin 2007] is a powerful and interesting example of this method. It tries to detect whether a new application $X$ is malicious or not, by running it in a system with FPT. Sensitive data, such as keystrokes that are unrelated to $X$ (e.g., a password you type in when logging to a remote machine) are tagged tainted. If at some point, any byte in the address space of $X$ is tainted, it means that the sensitive data has leaked into $X$, which should not happen. Thus, the program must be malicious and is probably a keylogger.

## 4. Test environment

To get a handle on the number of false positives, we track the spread of taint through the system for increasingly sophisticated versions of pointer tainting. The idea is that we mark sensitive data as tainted and monitor taint propagation over the system. If taint spreads to benign applications that should never receive tainted data, we mark it as a false positive.

For the experiments we use Qemu 0.9 [Bellard 2005] with vanilla Ubuntu 8.04.1 with Linux kernel 2.6.24-19-386 and Windows XP SP2. Depending on the test, we modified the Qemu emulator to taint either all characters typed at the keyboard, or all network data. We then propagate the taint via pointer tainting (using rules 1, 2, 3, and either 4a or 4b). Whether network or keyboard is tainted will be clarified when we discuss our experiments. The taint tag is a 32-bit value, so that each key stroke or network byte can have a unique colour, which helps in tracking the individual bytes.

To measure the spread of taint we repeatedly inspect the taintedness of registers at context-switch time. Tainted registers in processes that do not expect tainted input indicate unwanted taint propagation. The more registers are tainted, the worse the problem. The situation is particularly serious if special-purpose registers like the stack pointer (esp) and the frame pointer (ebp) are tainted. Once either of these registers is tainted, data on the stack also gets tainted quickly. Indeed, many accesses are made by dereferencing an address relative to the value of esp or ebp.

The measurements are conservative in the sense that even if the registers are clean at some point, there may still be tainted bytes in the process' address space. Moreover, we only check taint in registers during context switch time, probably again underestimating processes' taintedness. Taint may also leak across the kernel-userspace boundary in other ways, e.g., when tainted bytes are memory mapped into a process' address space. In other words, the real situation may be worse than what we sketch here. However the conservative approach we have implemented is sufficient to present the severity of the problem of false positives.

Context switches in Linux occur at just one well-defined point: the `schedule()` function. The scheduler is called either directly by a blocking call that will lead to a call to `schedule()` in the kernel (a voluntary context switch), or by interrupts and exceptions (a forced switch). For instance, a timer interrupt handler discovers that a process has used up its quantum of CPU time and sets a flag of the current process to indicate that it needs a reschedule. Just prior to the resumption of the user space process, this flag is checked and if it is set, the `schedule()` function is called.

The two methods differ in the way registers are saved. In particular, the general purpose x86 registers `eax`, `ecx` and `edx` are not saved on the call to `schedule()` on the voluntary context switch. The calling context is responsible for saving the registers are restoring them later. On interrupts and exceptions, all registers are saved at well defined points. The implication is that on voluntary switches, when we measure the state of the registers on return from `schedule()`, we ignore the taintedness of the above three registers. Whether they are tainted or not is irrelevant, as they will be overwritten later anyway. On a forced switch, when we inspect the condition of the process on the return from interrupt/exception handler, we look at all the registers. Summarizing, in any case the state of the registers being presented is captured once the original values are restored after the context switch. That reflects the state of processes rather than the state of kernel structures.

For a complete picture we also monitor the taintedness inside the kernel, during the `context_switch()` function.

As we cannot perform detailed analysis of Windows, we measure the state of the registers whenever the value of the `cr3` register changes. This x86 register contains the physical address of the top-level page directory and a change indicates that a new process is scheduled. For user mode processes the measurement is performed once the processor is operating in user mode. This way we are sure that we present the state of the process, and not some kernel structures used to complete the context switch.

## 5. Problems with pointer tainting

When we started implementing a keylogger detector by means of pointer tainting, we observed that taint spread rapidly through the system. We analyse now the problem of taint explosion both experimentally (Sections 5.1 and 5.2) and analytically (Section 5.3).

### 5.1 False positives in LPT

To confirm the immediate spread of taint in limited pointer tainting (LPT), we used the emulator that taints data coming from the network. Both for Linux and Windows alerts were quickly raised for benign actions like configuring the machine's IP address.

This is wrong, but not unexpected. We have already seen the causes in the LPT discussion in Section 3.2: without appropriate containment mechanisms, LPT propagates taint when combining an untainted base pointer and a tainted index and dereferencing such an address triggers an alert.

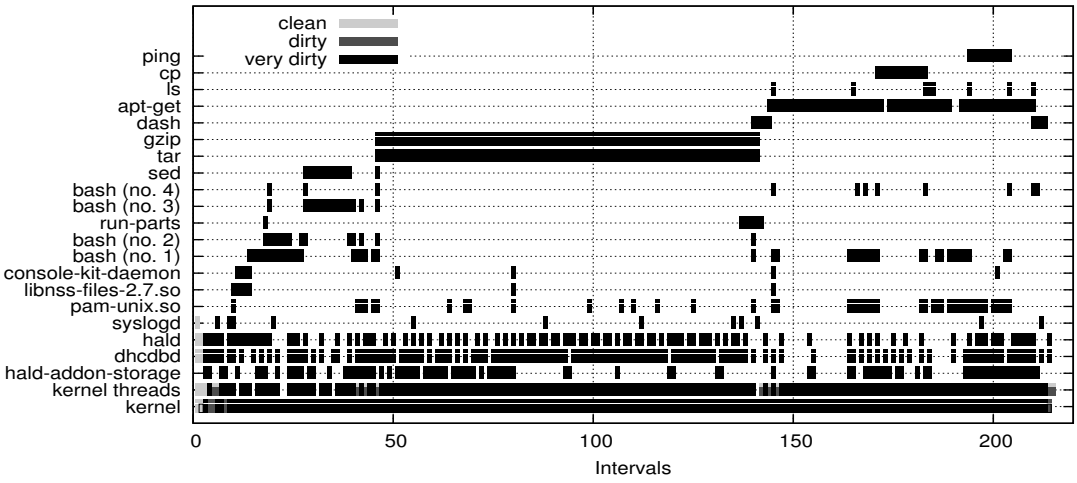


**Figure 2.** The taintedness of the processes constituting 90% of all context switches. In this and all similar plots the following explanation holds. The x-axis is divided into scheduling intervals, spanning 50 scheduling operations each. Time starts when taint is introduced in the system. In an interval, several processes are scheduled. For each of these, we take a random sample from the interval to form a datapoint. So, even if `gzip` is scheduled multiple times in an interval, it has only one datapoint. A datapoint consists of two small boxes drawn on top of each other, separated by a thin white line. The smaller one at the top represents the taintedness of ebp and esp. The bottom, slightly larger one represents all other registers. We use three colours: lightgrey means the registers are clean, darkgrey means less than half of considered registers are tainted, and black means that half or more are tainted (very dirty). Absence of a box means the process was not scheduled.

This is exactly what happened in our experiment. We discuss ways of addressing this problem in Section 6.2.

### 5.2 Taint explosion for FPT

To evaluate the spread of taint in full pointer tainting, we introduce a minimal amount of (tainted) sensitive information, and observe its propagation. After booting the OS, we switch on keystroke tracking (which taints keystroke data), and invoke a simple C program, which reads a user typed character from the command line. This is all the taint that is introduced in the system. Afterwards we run various applications, but do so using `ssh`, so no additional taint is added.

Figure 2 shows how taint spreads to the kernel and some of the most frequently scheduled processes. Aside from a few boxes on the very left, almost all applications and the kernel have at least half of the considered registers and `ebp` and `esp` tainted. Clearly, taint spreads very rapidly to all parts of the OS. Moreover in both this and the remaining experiments, `tar` and `gzip` should be completely clean as we use a bash script hardcoding the input and output filenames.

Figure 3 shows a similar picture for Windows XP. Here, performing simple tasks, we provide the guest operating system with new tainted keystrokes during the whole experiment. In more detail, we first launch the kernel debugger, `kd.exe`, and next switch on keystroke tagging. Thus, from this point onward data typed by the user is considered tainted. Next, we launch Internet Explorer, `IEXPLORE.exe`, and calculator, `calc.exe`. We perform simple web browsing, thus delivering tainted data to the Internet Explorer process. However, we do not provide the calculator with any typed characters, but we use solely the mouse. Finally, we switch off keystroke tagging, and consult the kernel debugger to dump values of the `cr3` register of running processes.

### 5.3 Analysis: the many faces of taint pollution

The above results show that pointer tainting without some containment measures is not useful. It is not possible to draw meaningful conclusions from the presence of taint in a certain process. A crucial step in the explosion of taint through the system is the pollution of the kernel. Data structures wholly unrelated to the keyboard activity pick up taint, and from the kernel, taint spills into user processes. As LPT simply raises an alert (and we have already seen how quickly this happens in a table lookup with tainted index), this section focuses on the more interesting case of FPT and we consider how taint spreads through the system in practice.

As mentioned earlier, incorrectly tainting `ebp` or `esp` is particularly bad, as accesses to local variables on the stack are made relative to `ebp`, and a 'pop' from the stack with a tainted `esp` will taint the destination. Unfortunately, the Linux kernel has numerous places where `ebp` and/or `esp` incorrectly pick up taint. Rather than discussing them all, we explain as an example how a common operation like opening a file, ends up tainting `ebp`, as well as various lists and structures in the kernel. The main purpose is to show that taint pollution occurs on common code paths (like opening files) and can be the result of complex interactions.

#### 5.3.1 Taint pollution by opening files - a case study

Taint pollution occurs due to calls to the `open()` system call in various ways. For the following analysis, we extended the emulator with code that logs the progression of taint through

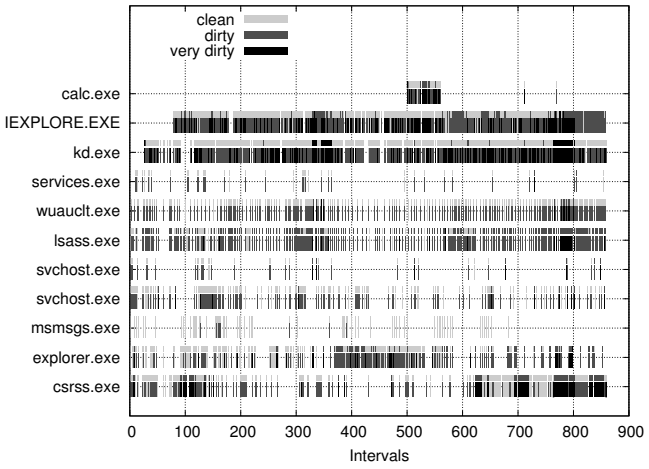


**Figure 3.** The taintedness of the processes constituting 95% of all context switches in Windows XP

```
[ 1] struct dentry * __d_lookup(struct dentry * parent,
[ 2]                            struct qstr * name)
[ 3] {
[ 4]     unsigned int hash = name->hash;
[ 5]     struct hlist_head *head = d_hash(parent,hash);
[ 6]     struct hlist_node *node;
[ 7]     struct dentry *dentry;
[ 8]
[ 9]     hlist_for_each_entry_rcu(dentry, node, head,
[10]                              d_hash) {
[11]       struct qstr *qstr;
[12]       ...
```

**Figure 4.** A snippet of the `__d_lookup()` function.

the system at fine granularity. We then manually analysed the propagation through the Linux source code by mapping the entries in the log onto the source.

The Linux Virtual Filesystem uses *dentry objects* to store information about the linking of a directory entry (a particular name of the file) with the corresponding file. Because reading a directory entry from disk and constructing the corresponding dentry object requires considerable time, Linux uses a *dentry cache* to keep in memory dentry objects that might be needed later. The dentry cache is implemented by means of a `dentry_hashtable`. Each element is a pointer to a list of dentries that hash to the same bucket and each dentry contains pointers to the next dentry in the list.

The real work in the `open()` system call is done by the `filp_open()` function which at some point accesses the dentry cache by means of the `__d_lookup()` function to search for the particular entry in the parent directory (see Figures 4 and 5). The second argument, `struct qstr* name`, provides the pathname to be resolved, where `name->name` and `name->len` contain the file name and length, and `name->hash` its hash value.

**Phase 1: taint enters kernel data structures.** To see how taint propagates incorrectly in `__d_lookup()`, let us assume that the filename was typed in by the user, so it is tainted. The hash is calculated by performing arithmetical and shift operations on the characters of the filename, which means that the hash in line 4 is tainted. The `d_hash()`

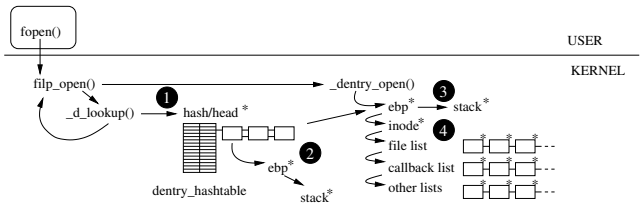


**Figure 5.** Taint pollution when a file is opened

function in line 5 first produces a new hash value from both the dentry object of the parent directory and the tainted hash of the filename (so that the new hash is also tainted) and then returns the head of the list of dentries hashing to this new hash value. This is the address of the element in the table with the index derived from the new hash value. The address is calculated as the table's base pointer plus tainted index and is therefore tainted. `head` in line 5 becomes tainted.

As is common in the Linux kernel, the singly linked list of dentries is constructed by adding a `struct hlist_node` field in a type that should be linked; in this case the dentry node. Each hlist_node field points to the hlist_node field of the next element, and a hlist_head field points to the start of the list. We iterate over the list (line 9), searching for the dentry matching the name, which will be found, if the file has been opened previously (which is quite common).

**Phase 2: ebp gets tainted.** During the iteration, `head` (and later `node`) contain pointers to the list's hlist_head and hlist_node link fields. Of course, these fields themselves are not interesting and the real work takes place on the associated dentry object. Therefore, the macro in line 9 performs a simple arithmetical operation to produce the address of the dentry, which results in tainting `dentry` (line 9). Worse, within the loop numerous checks of `dentry` are performed, and for efficiency, the `ebp` register is loaded with `dentry`'s tainted address. The result is that numerous values on the stack become tainted, and taint explosion is hard to avoid.

**Phase 3: ebp is cleaned and then tainted again.** By the time `__d_lookup()` returns, `ebp` is clean again, but taint keeps spreading. Now that the dentry object is found in the cache, the `filp_open()` function calls `__dentry_open()`, passing to it the tainted address of the dentry object. This function almost immediately loads the `ebp` register with the tainted address of the received dentry object. As a result, taint spreads rapidly through the kernel's address space.

**Phase 4: pollution of other structures via lists.** Taint spreads further across the kernel by dint of pointer arithmetic prevalent in structures and list handling. Linked lists are especially susceptible to pollution.

When we read a field of a structure pointed to by a tainted address, the result is tainted. Similarly, when we insert a tainted element `elem` to a list `list`, we immediately taint the references of the adjacent nodes. Indeed, the insertion operation usually executes the assignment `list->next=elem` which taints the `next` link. If we perform a list search or traversal, then the pointer to the currently examined ele-

ment is calculated in the following fashion: (1) `curr=list`, (2) `curr=curr->next`, and so the taintedness is passed on from one element to another.

If a list element is removed from one list and entered into another, the second list will also be tainted. For instance, if a block of data pointed to by a tainted pointer is freed, free lists may become tainted. By means of common pointer handling, the pollution quickly reaches numerous structures that are unrelated to the sensitive information.

Let us continue the example of opening files. As explained earlier, the `__dentry_open()` function is provided with the tainted address of the dentry object. This function executes the instruction `inode=dentry->d_inode` to determine the address of the inode object associated with `dentry`. The assignment taints `inode` as its value is loaded from the tainted address `dentry` plus an offset. Next, once the new file object `file` is initialised, we execute `head=inode->i_sb->s_files` as we insert the file into the list of opened files pointed to by `head` (`i_sb` is a field of the filesystem's superblock), so the head is tainted. As a result, the `file` insert operation immediately taints the references of the adjacent nodes in the list.

Finally, when the kernel has finished using the file object, it uses the `fput()` function to remove the object from the superblock's list and release it to the slab allocator. Without going into detail, we mention that dentry cache look-ups are lockless read-copy-update (RCU) accesses and that, as a result, the file objects end up being added to the RCU's per-CPU list of callbacks to really free objects when it is safe to do so. The list picks up the taint and, when it is traversed, spreads it across all entries in the list. The callback is responsible for releasing the tainted object to the slab.

#### 5.3.2 False positives and root causes of pollution

It is clear that due to false positives, limited pointer tainting and full pointer tainting in their naive, pure forms are impractical for automatically detecting memory corruption attacks and sensitive information theft, respectively. We have seen that taint leaks occur in many places, even on a common code path like that of opening a file. The interesting question is what the root causes of the leaks are, or phrased differently, whether these leaks have something in common.

After manually inspecting many relevant parts of the Linux kernel, we found three primary underlying causes for taint pollution. First, the tainting of `ebp` and `esp`. These pointers are constantly dereferenced and once they are tainted, LPT raises alerts very quickly, while FPT spreads taint almost indiscriminately as the stack becomes tainted.

Second, not all pointers are tainted in the same way and not all should propagate taint when dereferenced. If `A` is a tainted address and `B` is an address calculated relative to `A` (e.g., `B=(A+0x4)`), then `B` will be tainted. However, in many cases it might be unreasonable to mark `*B` as tainted. For example, let's assume that tainted A points to a field of a structure, `file_name`. Next, `B` is derived to hold the base address of this structure, `B = A-offset(file_name)`, and `B` becomes tainted. Now, depending on a security policy, we may or may not wish to mark `B->file_handler` as tainted. However, if all these structures are organized in a list, we certainly do not want to propagate taintedness to the next element of a list, `B->next`. On the other hand, if a pointer is itself calculated using tainted data (`C=A+eax`, where `eax` is tainted), the taint *should* be propagated, as C might be pointing to a field in a translation table. Notice that all these cases are hard to distinguish for emulators or hardware.

Third, if pointer tainting is applied only for detecting memory corruption attacks on non-control data, rather than tracking keystrokes and other sensitive data, taint may leak due to table lookups, as discussed in Section 3.2.

#### 5.3.3 False negatives: is pointer tainting enough?

While false positives are more serious than false negatives for automatic detection tools, a system that misses most of the attacks is not very useful either. 'Pure' pointer tainting in LPT or FPT does not have many false negatives, but even without any containment of taint propagation, pointer tainting does not detect all the attacks it is designed for. For instance, LPT will detect modification of non-control data by means of a format string attack, or a heap overflow that corrupts freelist link pointers. However, it will miss modification of non-control data by means of a direct buffer overflow. Limited mitigation may be possible by checking system call arguments, as is done in Vigilante [Costa 2005], but the fundamental problem remains. Consider for instance, a buffer on the heap or stack that contains the username and may be located in memory just below a field that indicates the user's privilege level. If attackers can overflow the buffer, they can modify the privilege level.

Similarly, FPT and LPT both miss implicit information flows. Implicit information flows have no direct assignment of a tainted value to a variable (which would be propagated by pointer tainting). Instead, the value of a variable is completely determined by the value of tainted data in a condition. For instance, if $x$ is tainted, then information flows to $y$ in the following code: `if (x=0) y=0; else y=1;`. As we do not track implicit information flows, false negatives are quite likely. This is particularly worrying if FPT is used to detect potential privacy-breaching malware, as it gives the untrusted code an opportunity to 'launder' taint. As pointed out by [Cavallaro 2008] purely dynamic techniques cannot detect implicit flows. The authors explain that it is necessary to reason about the assignments that take place on the unexecuted program branches, and also provide a number of reasons making the problem intractable for x86 binaries.

If we are to have any hope at all of catching sophisticated privacy-breaching malware with FPT, we need to detect and raise an alert immediately when taint enters the untrusted code, lest it be laundered. As soon as untrusted code is allowed to run it can trick the system into cleaning the data.

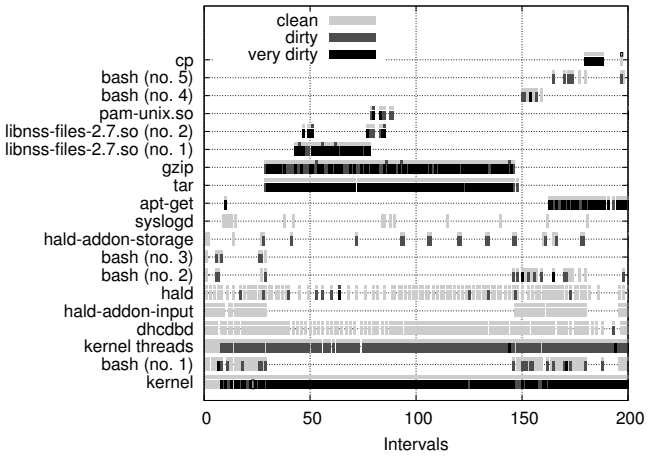


**Figure 6.** Taint pollution with `esp`/`ebp` protection for code involved in 90% of the context switches (Linux)

Like most work on pointer tainting, we assume that false negatives in pure pointer tainting are not the most important problem. However, to deal with the false positives, we are forced to contain taint propagation in various ways. Doing so will reduce the false positive ratio, but the opportunities for false negatives will increase significantly.

## 6. Containment techniques

We have seen that without containment, pointer tainting is not usable. We now evaluate ways to control the spreading.

### 6.1 Containment for LPT *and* FPT: ebp/esp protection

The first cause of pollution in FPT (and false positives in LPT) mentioned in Section 5.3.2 is tainting of `esp` and `ebp`. We can simply remove it with minimal overhead by never applying pointer tainting to tainted values of `ebp` or `esp`. However, on occasion `ebp` is also used as a temporary general purpose register. Having analysed a number of scenarios that involved a tainted `ebp`, we devised a simple heuristic, and clean `ebp` whenever its value is big enough to serve as a frame pointer on the stack. Doing so introduces false negatives into the system in case `ebp` is used as a temporary general purpose register and serves as a pointer. However, we expect this to be rare.

We implemented the above restriction in our emulator and again evaluated the spread of taint through the system. The results for Linux, shown in Figure 6, indicate that while the spread has slowed down a little compared to Figure 2, taint still propagates quickly. Observe that `ebp` is still tainted occasionally. This is correct. It means that `ebp` was used as a container. For lack of space, we do not show the plot for Windows. We will show a combined plot later (Figure 9).

### 6.2 LPT-specific containment techniques

The most important cause of false positives in LPT involves taint pollution via table lookups. As LPT uses pointer tainting only to detect memory corruption attacks, rather than tracking sensitive data, we should try to prevent taint from leaking due to lookups. Proposed solutions revolve around detecting some specific pointer arithmetic operations [Suh 2004], bounds-checking operations [Chen 2005a, Dalton 2007], and more recently pointer-injection [Katsunuma 2006, Dalton 2008]. Because of conversion tables, none of these techniques are suitable for FPT.

**Detecting and sanitising table accesses.**

Suh et al. [Suh 2004] sanitise table lookups even when the index is tainted and assume that the application has already performed the necessary bounds checks. The method is impractical as it requires us to recognise table lookups, while many architectures, including the popular x86, do not have separate instructions for pointer arithmetic. On x86, we can only instrument those instructions that calculate an address from base and index. Then we propagate the taint of the base and skip that of the index. However, the use of `add` and `mov` instructions to calculate pointers is extremely common in real-world code and these cannot be monitored in the same way. As a result, this method leads to many false positives. Others have pointed out that this policy is also prone to false negatives in case of return-to-libc attacks [Dalton 2006].

**Detecting bounds checks**

Chen et al. [Chen 2005a] argue that most table lookups are safe even if the index is tainted, as long as the index was properly bounds-checked. Thus, to reduce false positives, we may try to detect bounds-checks at runtime, and drop the operand's taint. Bounds-checks are identified by a `cmp` instruction of the index with an untainted operand. As the `and` instruction is also frequently used for bound checks [Dalton 2007], we also clean the first source operand of `and` if the second operand is clean and has a value of $2^n - 1$.

While simple and fast, the method suffers from false positives and negatives, some of which were noted by others [Dalton 2006; 2007; 2008]. We are the first to find the last 2 in the list below.

In many conversion tables, a lookup simply returns a different representation of the input and cleaning the tag leads to false negatives. For instance, the taintedness of suspicious input is dropped as it passes through translation tables, even if the data is then used for a buffer overflow or other exploit. Incorrectly dropping taint in a way that can be exploited by attackers is known as taint laundering (Section 5.3.3). False negatives also occur when the `cmp` and `and` instructions are used for purposes other than bounds checking.

In addition, the method is prone to false positives if code does not apply bounds-checking at all or uses different instructions to do so. Many lookups take place with an 8-bit index in a 256-entry table and are safe without bounds check.

Furthermore, taint often leaks to pointers in subtle ways that are not malicious at all. For instance, many protocols have data fields accompanied by length fields that indicate how many bytes are in the data field. The length may be

used to calculate a pointer to the next field in the message. A subtle leak occurs when the length *is* bounds checked, but the check is against a value that is itself tainted. For instance, a check whether the length field is shorter than IP's total length field (minus the length of other headers). A comparison with tainted data does not clean the index.

Yet another way for taint to escape and cause false positives, is when check and usage of the index are decoupled. For instance, the index is loaded into a register from memory and checked, which leads us to clean the register. However, by the time the index is actually used, it may well have to be loaded from the original (tainted) memory address again, because the register was reused in the meantime. This again leads to a tainted dereference and thus an alert. We see that for various reasons, raising alerts immediately on tainted dereferences is likely to trigger many false positives.

We have just discussed why current solutions that revolve around detecting bounds checks and table accesses are insufficient and incur both false positives and false negatives. We implemented these policies, and experiments performed on the emulator confirm our objections: control flow diversions were reported instantly. In addition, on architectures like x86, there is little distinction between registers used as indices, normal addresses, and normal scalars. Worse, the instructions to manipulate them are essentially also the same. As a result this problem is very hard to fix, unless a particular coding style is enforced.

**Pointer injection detection**

This brings us to recent and more promising work which prevents memory corruption attack by detecting when a pointer is injected by untrusted sources [Katsunuma 2006]. The most practical of these, Raksha [Dalton 2008] identifies valid pointers, which it marks with a $P$ bit, and triggers an alert if a tainted pointer is dereferenced that is not (derived from) a valid pointer.

For this purpose, it scans the data and code segments in ELF binaries to map out in advance all legitimate pointers to statically allocated memory and marks these with a $P$ bit. To do so, it has to rely on properties of the SPARC v8 architecture that always uses two specific instructions to construct a pointer and has regular instruction format. In addition, it modifies the Linux kernel to also mark pointers returned by system calls that allocate memory dynamically (`mmap`, `brk`, `shmat`, etc) with a $P$ bit. Furthermore, as the kernel sometimes legitimately dereferences untrusted pointers it uses knowledge of SPARC Linux to identify the heap and memory map regions that may be indexed by untrusted information (and uses the kernel header files to find the start and end addresses of these regions).

The method effectively stops false positives, but false negatives are possible. For instance, it is possible to overflow a buffer to modify an index that is later added to a legitimate address. The resulting pointer would have the P bit set and therefore a dereference would not trigger alerts.

More worrying is that the method is very closely tied to the Linux/SPARC combination and portability is a problem. For instance, it would not work well on x86 processors running Windows. First, x86 makes it much harder to detect pointers to statically allocated memory. Second, we cannot modify the kernel, so that we are forced to add specific handling for several system calls in hardware or emulator (and the number of system calls in Windows is large). Third, we cannot identify kernel regions that may be indexed with untrusted data. To err on the safe side, we would have to assume that certain data values are pointers when really they are not, and that the entire kernel address space could be pointed to by untrusted data. As a result, we expect many false negatives on x86. Even so, while limited to OS/architecture combinations similar to Linux/SPARC, Raksha is the most reliable LPT implementation we have seen.

### 6.3 FPT-specific techniques

Section 5.3.2 identified primary causes for pollution. We now try to remove them without crippling the method.

**White lists and black lists**

The simplest solution is to whitelist all places in the code where taint should be propagated using pointer tainting, or alternatively, blacklist all places where it should not be propagated. Neither scheme works in practice. Whitelisting is impossible unless you know all software in advance (including the userspace programs) and well enough to know where taint should propagate. This is certainly not the case when you are monitoring potential malware (e.g., to see if it is a keylogger). It is also difficult for large software packages (say, OpenOffice, or MS Word), or any proprietary code. Finally, whitelisting only a small subset of the places reduces FPT to taint analysis with a minimal extension.

Blacklisting also suffers from the problem that you have no detailed knowledge over all programs running on your system. In addition, the number of taint leaks is enormous and blacklisting them all is probably not feasible. Notice that even if we managed to blacklist part of the software, including the Linux kernel and a few applications, for instance, that still would not be enough. Assume that one of the programs we do not blacklist causes unrelated data to be tainted. Next, if such data is communicated to other processes, they become tainted, and a false alarm is raised. Such unrelated tainted data can enter kernel structures during system calls.

Finally, blacklisting and whitelisting both have a significant impact on performance. Thus, we do not consider whitelisting or blacklisting a feasible path to remedy FPT.

**Landmarking**

Easy fixes like ebp/esp protection and white/black-listing do not work. In this section, we discuss a more elaborate scheme, known as *landmarking*, that contains taint much more aggressively. Unfortunately, as a side effect, it significantly reduces the power of pointer tainting which leads to

```
[1] typedef struct test_t {
[2]   int i;
[3]   struct test_t* next;
[4] } test_t, *ptest_t;
[5]
[6] ptest_t table[256] = ...;   // initialised
[7] ptest_t i1 = table[index];  // tainted
[8] ptest_f i2 = i1->next;      // clean
[9] int     i3 = i1->i;         // clean
```

**Figure 7.** An example of landmarking.

many false negatives. In addition, it still incurs false positives and significantly increases the runtime overhead. Nevertheless, this is the most powerful technique for preventing taint explosion we know. A similar technique appears to have been used in Panorama [Yin 2007], but as our landmarking is slightly more aggressive and, hence, should incur fewer false positives, we will limit the discussion to landmarking.

Recall that the second primary cause of unwanted taint propagation is due to pointers being relative to a tainted address: if `A` is a tainted address, and an address `B` is calculated relative to `A` (e.g., `B=A+0x4`), then `B` is tainted as well, even though tainting `*B` is often incorrect. As a remedy, we will let `B` influence the taintedness of `*B` only if it *itself* was calculated using tainted data. So, with `A` and `eax` tainted, we will exclude `B=A+0x4` from taint propagation, but keep `C=A+eax`.

For this purpose, we introduce *landmarks*. Landmarks indicate that an address is 'ready to be used for a dereference'. We have reached a landmark for `A`, if all tainted operations up to this point were aimed at calculating `A`, but not a future value of `B` derived from `A`. Rephrasing, as soon as a value is a landmark (and thus a valid and useful) address, dereferences should propagate taint. However, values derived from the landmark have to be modified with tainted data again in order to make the derived value also qualify for pointer taintedness. Thus, we limit the number of times a tainted value can influence the taintedness of a memory access operation.

In practical terms, we say that a value forms a complete and useful address only when it is used as such. In other words, we identify landmarks either by means of a dereference, or by an operation explicitly calculating an address, such as the `lea` instruction on x86 that calculates the effective address and stores it in a register.

***Example*** Consider the code snippet shown in Figure 7. We access the second item of a list rooted at `table[index]`, where the index is assumed to be tainted. First, in line 7, the pointer to the head of the list is fetched from the table. To calculate the memory load address, `(table + index*8)`, we use a tainted operand, `index`, which has never been dereferenced before, and so `i1` becomes tainted. However, we have just reached the landmark for `i1`, meaning that dereferences of `i1` propagate taintedness, but addresses derived from `i1` in a clean way do not. Next, in the second assignment, line 8, we access memory at the address calculated by increasing `i1` by 4, a clean constant. Thus `(i1 + 4)` when dereferenced does not propagate taintedness, and `i2` is clean. A similar reasoning holds for clean `i3`. Based on this example one might think that landmarking solves our problems, as we propagate the taintedness to the elements of a (translation) table, but we do not spread it over elements of a list.

***Problems with landmarking*** Unfortunately, landmarking is not just a rather elaborate technique, it also cripples the power of pointer tainting and opportunities for false negatives abound.

Assume that `p` is a pointer whose calculation involves a tainted operand, and we load values `v0=*p` and `v1=*(p+1)` from memory. In a first possible scenario, the compiler translates the code such that `p` is calculated once for both load operations. In that case, the first of the loaded variables becomes tainted, and the other one is clean. So, depending on the order of instructions, `v0=*p; v1=*(p+1);` vs. `v1=*(p+1); v0=*p;`, we get different results. This is strange. On the other hand, if the compiler translates the code such that `p` is calculated *twice*, once for each of the variables, then both values are tainted. Such inconsistent behaviour makes it hard to draw conclusions based on the results of landmarked pointer tainting. Moreover, it clearly introduces false negatives.

Another example of false negatives stems from translation tables containing structures instead of single fields. Let's refer to Fig 7 once more, where `i1` (line 7), is tainted, but `i3=i1->i` (line 9), is clean. Now imagine that the `test_t` structure contains various representations of characters, say ASCII and EBCDIC. In that case, the table access makes us lose track of sensitive data, which is clearly undesirable.

This weakness can also be exploited to cause leakage of secret data. Assume that a server receives a string-based request from the network and returns a field from a `struct X`, pointed to by `xptr`. If an attacker is able to modify `xptr` (for instance, by overflowing the request buffer with a long string), then the server returns the contents of `xptr->field_offset` which can point to an arbitrary place in the memory. For the same reasons as in the example above, the result will be clean.

The *best* thing about landmarking is that we contain the spread of taint very aggressively and it really is much harder for taint to leak out. The hope is that, in combination with `ebp`/`esp` protection, landmarking can stop the pollution, so that (an admittedly reduced version of) FPT can be used for automatic detection of keyloggers.

The *worst* thing about landmarking is that it does not work. It still offers ample opportunities for false positives. This is no surprise, because even if we restrict taint propagation via pointer tainting in one register, nothing prevents one from copying the register, perhaps even before the dereference, and adding a constant value to the new register. As the new register was not yet used for dereferencing, taint will be (incorrectly) propagated.

Another possible reason for false positives arises when programs calculate directly the address of an element (or

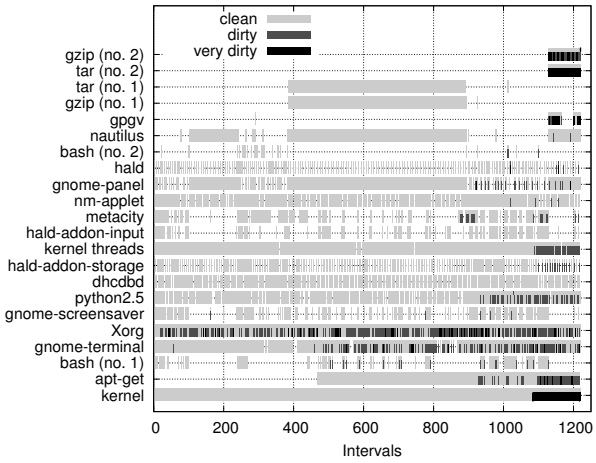


**Figure 8.** Pollution with landmarking and ebp/esp protection combined for code involved in 90% of the switches (Linux)

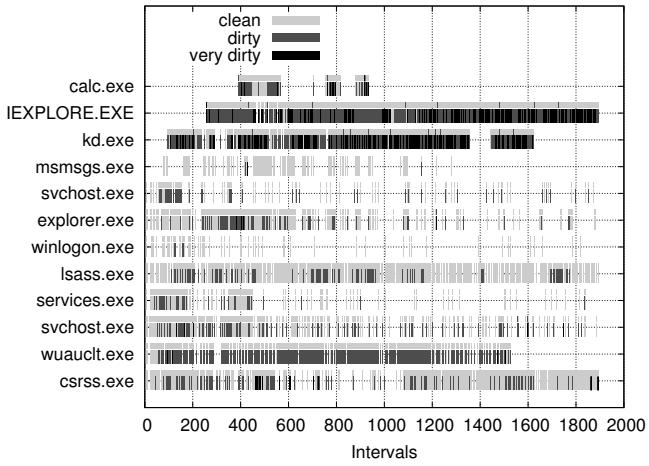


**Figure 9.** Taintedness of processes constituting 95% of all context switches in XP with landmarking and ebp/esp protection

field within a struct, without going through an immediate pointer. Consider an array `A` of `struct{int a;int b;}` and assume the index is tainted. If we first calculate the address of `A[index]` and use this to calculate the address of the field `b`, everything will be fine. No taint is unduly propagated to `b`. However, if we directly calculate a pointer to `b` (e.g., `int *p = (char*)A+8*index+4;` we would propagate taint incorrectly to 'b'. The array example is very simplistic and perhaps a bit contrived but the same (very real) problem may hold for queues, stacks, and hashtables.

We implemented landmarking and analysed the spread of taint when applying it together with `ebp/esp` protection. The results are shown in Figure 8. Taint pollution takes considerably longer than with just `ebp/esp` protection. Some of the processes that receive taint early (like the Xorg X server, or the screensaver), conceivably should have access to the tainted bytes. However, after some time taint again spreads to completely unrelated user processes (e.g., tar, nautilus, hald, python, apt-get, etc.), as well as to the kernel and kernel threads. The results for Windows (Figure 9) are even worse. Notice that *all* processes are occasionally tainted. The `calc.exe` process, for instance, should not get any taint at all, as we provide input using mouse, and not keyboard. `wuauclt.exe` is the AutoUpdate Client of Windows Update and is used to check for available updates from Microsoft Update, and thus it's not expected to process keyboard events either.

## 7. How bad are things?

We conclude with an overall assessment of FPT and LPT.

### 7.1 FPT on current hardware is fundamentally broken

We have discussed a few solutions to contain the spurious taint propagation, but they are prone to false negatives, and only slow down the outburst of false positives. However, the problem is even more serious as there are undecidable cases when the (most common) hardware itself is not able to firmly establish the taintedness of a load operation. For instance, it may be impossible to distinguish accesses to translation tables from accesses to `next` fields in a linked list. We discuss such a scenario below and argue that without an external oracle (like *a priori* annotated translation tables or support from a source code analyzer) we are not able to successfully apply FPT. We believe that on current hardware, FPT with current containment techniques is not feasible.

Assume that `kbd_data` is a tainted keystroke and that an application needs to find a lower case value associated with that keystroke. Clearly, we want to propagate taintedness to this derived value. The following code (similar to the GNU C library (`glibc-2.7`) `to_lower()` function) is used to obtain the lower case value:

```
[1] attributes = transl_table[kbd_data];
[2] lower_case = attributes->lower;
```

In line 1, to get the address of `attributes`, a non-tainted pointer (`transl_table`) is combined with a tainted index, `kbd_data` and the pointer is dereferenced. In other words, atttributes is tainted. In line 2, this new tainted pointer is updated with a constant, and dereferenced to load the `lower_case` value. Unfortunately, taint is not propagated.

At the same time there are numerous cases where the emulator/hardware is executing a similar sequence of instructions where results should *not* be tainted. Consider the following code, which we already presented in Section 5.3.1

```
[1] struct hlist_head *head = d_hash(parent, hash);
[2] struct dentry *dentry = head->first;
[3] /* (...) */
[4] dentry = dentry->next;
```

Line 1 again combines a non-tainted pointer (the address of the hashtable) with a tainted index derived from `hash`. Next, in line 2 (and in line 4), the newly derived tainted pointer, `head` (or `dentry` in line 4), is again altered using a constant index to further fetch a next entry from the list. An equivalent sequence of instructions as in the previous example should now *not* propagate taint. Since, we have

argued that black/white listing cannot solve the problem fully, false positives would be unavoidable.

We do not believe that hardware can distinguish the two cases without an external oracle. Of course, we may throw more heuristics at the problem. For instance, we could apply landmarking such that it allows for dereferences of one level deep, so that while `dentry` is tainted, a dereference does not propagate taint. However, the opportunity for false positives would increase. Moreover, since we may have structures with multiple levels of nesting, it is hard to see where we should draw the line.

### 7.2 Challenges for LPT on popular hardware

As we saw in Section 6.2, pointer injection detection seems to be a promising technique for containing taint propagation. If we can get it to work on commonly used hardware, LPT may be a powerful technique for detecting attacks against non-control data. Unfortunately, it seems impossible to achieve this unless we are able to recognise existing system pointers reliably. [Dalton 2008] shows that this is possible for Linux on the SPARC by dent of architectural 'features' (e.g., two specific instructions are used to form a pointer). It is an open challenge to do something similar on x86-like architectures, *or* to invent completely new techniques for containtment of taint. Meeting this challenge means that we salvage an important technique for detecting non-control data attacks. Until that time, we think pointer tainting has serious problems that prevent it from being used in real systems.

## 8. Related work

The work on dynamic information flow tracking (DIFT) by [Denning 1977] forms the basis of taint analysis with pointer tainting. The paper describes a way of verifying the secure flow of information through a program. DIFT is applied in hardware by Suh et al. [Suh 2004]. While taint is propagated whenever memory is accessed using tainted pointers, the system triggers alerts only for control flow diversions.

The technique of pointer tainting for non-control data attack detection was formally introduced by [Chen 2004], and later evaluated in a hardware design [Chen 2005a;b]. Any dereference of tainted pointer triggers an alert and the technique is evaluated with a number of attacks, and six SPEC2000 applications. No false positives are reported.

[Dalton 2006] points out that naive propagation rules that trigger an alert when a pointer dereference has any of its source operands tainted are problematic. Indexing an array with a tainted index in many cases need not be unsafe if it was properly bounds-checked. The proposed solution is to propagate only the taint bit of the base pointer. While DIFT [Suh 2004] optimistically assumes that bounds checking is always done by the applications, this is clearly not true in practice. The authors suggest to apply untainting only for instructions dedicated to input validation which requires an architecture that can distinguish such instructions. In addition, they indicate several weaknesses in [Chen 2005a], most notably its inability to deal with translation tables.

Raksha [Dalton 2007] is yet another hardware (FPGA-based) approach with support for pointer tainting. However, the system is quite flexible and able to handle various DIFT models besides pointer tainting.

Panorama [Yin 2007] differs from the previous projects in that it is designed solely to check whether sensitive data leaks into software that may or may not be malicious. All data from sensitive source is marked tainted and the potential malware is run in a system under pointer tainting. The program under scrutiny is considered suspicious if it processes tainted data. [Egele 2007] is a very similar approach, but focused on detecting spyware in a web browser. HookFinder [Yin 2008] wants to determine whether a piece of malicious code has implanted a hook in the OS. Dynamic taint analysis, including full pointer tainting, is used to track what they refer to as impacts on the OS made by the untrusted software, and checking whether they exhibit a desired hooking behaviour.

[Xu 2006] presents a dynamic taint analysis technique to detect input validation attacks, implemented as a source-to-source transformation. For fear of false positives the authors do not track dereferences, only the direct array access is supported.

## 9. Conclusion

We have analysed pointer tainting, considered one of the most powerful techniques to detect keyloggers and memory corruption attacks on non-control data. Both in the analysis and in experiments, the method proved problematic due to the large number of false positives, even when we apply all methods that we know of for containing the spread of taint. We argued that full pointer tainting is probably not suited for detecting privacy-breaching malware like keyloggers. Moreover, it is even unclear whether limited pointer tainting can be applied to detect automatically memory corruption attacks on the most popular PC architecture (x86) and the most popular OS (Windows).

## Acknowledgements

This work is sponsored by the EU FP7 WOMBAT and the EU FP7 FORWARD projects. We are grateful to Christopher Kruegel, Mike Dalton, and Heng Yin for clarifying the way in which they addressed the problem of taint explosion in their work. Finally, we thank Andrew Warfield (our shepherd), Rebecca Isaacs, Manuel Costa, and the (excellent) anonymous EUROSYS reviewers for their feedback on earlier versions of this paper.

## References

[Akritidis 2008] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *SP '08: 2008 IEEE Symposium on Security and Privacy*, 2008.

[Bellard 2005] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: 2005 USENIX Annual Technical Conference*, 2005.

[Bhatkar 2005] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *SSYM'05: 14th USENIX Security Symposium*, 2005.

[Castro 2006] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: 7th symposium on Operating systems design and implementation*, 2006.

[Cavallaro 2008] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA '08: 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[Chen 2004] S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Formal reasoning of various categories of widely exploited security vulnerabilities using pointer taintedness semantics. In *Proc. of IFIP SEC*, 2004.

[Chen 2005a] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and I. Ravishankar. Defeating memory corruption attacks via pointer taintedness detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.

[Chen 2005b] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *SSYM'05: 14th USENIX Security Symposium*, 2005.

[Costa 2005] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[Cowan 1998] C. Cowan, C. Pu, D. Maier, H. Hintony, Walpole J., P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, 1998.

[Crandall 2004] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th Interational Symposium on Microarchitecture*, 2004.

[Dalton 2006] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing hardware architectures for security. In *5th Workshop on Duplicating, Deconstructing, and Debunking*, 2006.

[Dalton 2007] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[Dalton 2008] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *SSYM'08: 17th Usenix Security Symposium*, 2008.

[Denning 1977] D. Denning and P. Denning. Certification of programs for secure information flow. *Commnic. ACM*, 20(7), 1977.

[Egele 2007] M. Egele, Ch. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *ATC'07: 2007 USENIX Annual Technical Conference*, 2007.

[Elphinstone 2007] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *HOTOS'07: 11th USENIX workshop on Hot topics in operating systems*, 2007.

[Giffin 2004] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *The 11th Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[Ho 2006] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys '06: 1st European Conference on Computer Systems*, 2006.

[Jim 2002] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX 2002 Annual Technical Conference*, 2002.

[Katsunuma 2006] S. Katsunuma, H. Kurita, R. Shioya, K. Shimizu, H. Irie, M. Goshima, and S. Sakai. Base address recognition with data flow tracking for injection attack detection. In *PRDC '06: 12th Pacific Rim International Symposium on Dependable Computing*, 2006.

[Newsome 2005] 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 (NDSS)*, 2005.

[Portokalidis 2008] G. Portokalidis and H. Bos. Eudaemon: Involuntary and on-demand emulation against zero-day exploits. In *EuroSys '08: 3rd European Conf. on Computer Systems*, 2008.

[Portokalidis 2006] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys '06: 1st European Conference on Computer Systems*, 2006.

[ProcessLibrary.com ] ProcessLibrary.com. zango.exe. `http://www.processlibrary.com/directory/files/zango/`.

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

[Raywood 2008] Dan Raywood. Sinowal trojan steals data from around 500,000 cards and accounts. *SC Magazine*, Oct 2008.

[Slowinska 2007] A. Slowinska and H. Bos. The age of data: pinpointing guilty bytes in polymorphic buffer overflows on heap or stack. In *ACSAC'07*, 2007.

[Suh 2004] E. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32(5):85–96, 2004.

[Venkataramani 2008] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *HPCA'08*, 2008.

[Xu 2006] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, 2006.

[Yin 2008] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[Yin 2007] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proc. of the 14th ACM conference on Computer and communications security*, 2007.