On the Limits of Information Flow Techniques for Malware Analysis and Containment*

Lorenzo Cavallaro[1], Prateek Saxena[2], and R. Sekar[3]

[1] Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano, Italy
[2] Computer Science Department
University of California at Berkeley, USA
[3] Computer Science Department
Stony Brook University, USA

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 difficulties that lead to gaping holes in defense. These holes arise not only due to the limitations of information flow analysis techniques, but also the nature of today's software architectures and distribution models. This paper highlights these problems using an array of simple but powerful evasion techniques that can easily defeat taint-tracking defenses. Given today's binary-based software distribution and deployment models, our results suggest that information flow techniques will be of limited use against future malware that has been designed with the intent of evading these defenses.

1 Introduction

Information flow analysis has long been recognized as an important technique for defending against attacks on confidentiality as well as integrity [6,8]. Over the past quarter century, information flow research has been concentrated on static analysis techniques, since they can detect *covert channels* (e.g., so-called implicit information flows) missed by runtime monitoring techniques.

Static analyses for information-flow have been developed in the context of high-level, type-safe languages, so they cannot be directly applied to the vast majority of COTS software that is available only in binary form. Worse, software obfuscation and encryption techniques commonly employed in malware (as well as some benign software for intellectual property protection) render any kind of static analysis very difficult, if not outright impossible. Even in the absence of obfuscation, binaries are notoriously hard to analyze: even the basic step of accurate disassembly does not have solutions that are robust enough to work on large x86 binaries. As a result, production-grade tools that operate on binaries rely on dynamic (rather than static) analyis and instrumentation [3,7,17,24,26].

Following this observations, several researchers have recently developed dynamic information-flow techniques for COTS binaries [10,15,29,30,36]. These techniques,

* This research is supported in part by an ONR grant N000140710928 and an NSF grant CNS-0627687, and was carried out while the first two authors were at Stony Brook University.

along with source-to-source based transformation approaches, have enabled accurate detection of a wide range of attacks on trusted software[4] including those based on memory corruption [15,36], format-string bugs, command or SQL injection [2,28,43], cross-site scripting [40], and so on. More recently, researchers have reported significant successes in applying dynamic information flow techniques on existing malware, both from the perspective of understanding their behavior [1], and detecting runtime violation of policies [13,34]. Although dynamic taint analysis technique is quite robust for protecting trusted software, its application to untrusted (and potentially malicious) software is subject to a slew of evasion techniques that significantly limit its utility. We point out that understanding the limitations of defensive techniques is not 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 [19] and W32/Ratos [38] 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 [18]) to make malware debugging sessions harder [37,39]. 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. We have organized our discussion into three major sections as follows, depending on the context in which information flow is being used.

Stand-alone malware. When applied to malware, a natural question is whether the covert channels that were ignored by dynamic techniques could be exploited by adaptive malware to thwart information-flow based defenses. These covert channels were ignored in the context of trusted software since their "capacity" was deemed too small to pose a significant threat. More importantly, attackers do not have any control over the code of trusted software, and hence cannot influence the presence or capacity of these channels. In contrast, malware writers can deliberately embed covert channels since they have complete control over malware code. In this paper, we first show that it is indeed very easy for malware writers to insert such covert channels into their software. These evasion techniques are simple enough that they can be incorporated manually, or using simple, automated program transformation techniques. We show that it is very difficult to defeat these evasion techniques, unless very conservative reasoning is employed, e.g., assuming that any information read by a program could leak to any of its outputs. Unfortunately, such weak assumptions can greatly limit the purposes to which dynamic information flow analysis can be used. For instance, Stinson *et al*. [34] use information flow analysis to detect "remote-control" behavior of bots, which is identified when arguments to security-critical system calls are tainted. If a conservative notion of tainting is used, then all programs that communicate over the network would have to be flagged as "bots," which would defeat the purpose of that analysis.

Malware plug-ins. Next, we consider recent evolution in software deployment models that has favored the use of plug-in based architechtures. Browser helper objects (BHOs), which constitute one of the most common forms of malware in existence to-

[4] In this paper, the term "trusted software" is used to refer to software that is trusted to be benign.

day, belong to this category. Other examples include document viewer plug-ins, media codecs, and so on. We describe several novel attacks that are possible in the context of plug-ins:

- *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 violates 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.

While conservative notions of tainting could potentially be used to thwart these attacks [33], this would restrict the applicability of information-flow techniques even more.

Analyzing future behavior of malware. 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 techniques that can reason about program paths that are not exercised during monitoring. While one may attempt to force execution of all program paths, such an approach is likely to be very expensive, and more likely to suffer from semantic inconsistencies that may arise due to forcing execution down branches that are not taken during execution. A more selective approach has been proposed by Moser *et al.* [1] that explores paths guarded by tainted data, rather than all paths. This technique has been quite successful in the context of existing malware. The heart of this approach is a technique that uses a decision procedure to discover memory locations that could become tainted as a result of program execution, and explores branches that are guarded by such data. In Section 4, we show that these trigger discovery mechanisms (and more generally, the technique for discovering which data items can become tainted) can be easily evaded by purposefully embedding memory errors in malicious code.

Paper organization. Sections 2 through 4 describe our evasion techniques, organized along the lines described above. Where possible, mitigation of these evasions and their implications on information flow analyses are discussed as well. A summary of related work is provided in Section 5, followed by concluding remarks in Section 6.

2 Stand-Alone Untrusted Applications

For the sake of concreteness, we discuss the impact of evasion attacks, as well as mitigation measures, in the context of the "remote control" behavior detection technique

presented by Stinson *et al.* [34], although the evasion techniques themselves are applicable against other defenses as well, e.g., dynamic spyware detection [13].

Stinson *et al.* observed that 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. Below, we describe simple evasion measures that allow malware to "drive a truck" through the gaps in most dynamic taint-tracking techniques, and proceed to discuss possible mitigation mechanisms and their implications.

2.1 Evasion using Control Dependence and Implicit Flows

Dynamic information flow techniques that operate on trusted software tend to focus on *explicit flows* that take place via assignments. It is well known that information can flow from a variable y to another variable x without any explicit assignments. Indeed, a number of covert channels for information flow have been identified by previous research in this area. We demonstrate the ease of constructing evasion attacks using these covert channels. We focus on two forms of non-explicit flow, namely, control dependences and implicit flows.

Control dependence arises when a variable is assigned within an if-then-else statement whose condition involves a sensitive (tainted[5]) variable, e.g.,

$$\textbf{if } (y = 1) \textbf{ then } x := 1; \textbf{ else } x := 0; \textbf{ endif}$$

Clearly, the value of x is dependent on y, even though there is no assignment of the latter to the former. In particular, the above code snippet enables copying of a single bit from y to x without using direct assignments between them. Using an n-way branch (e.g., a switch statement with n cases) will allow copying of $\log n$ bits. A malware writer can propagate an arbitrarily large amount of information without using explicit flows by simply enclosing the above code snippet within a loop.

Implicit flows arise by virtue of semantic relationships that exist between the values of variables in a program. As an example, consider the following code snippet that allows copying of one bit of data from a sensitive variable y to w without using explicit flows or control dependences:

1. $x := 0;\ z := 0;$
2. **if** $(y = 1)$ **then** $x := 1;$ **else** $z := 1;$ **endif**
3. **if** $(x = 0)$ **then** $w := 0;$ **endif**
4. **if** $(z = 0)$ **then** $w := 1;$ **endif**

[5] Typically, the term "taint" is used in the context of integrity, while "sensitive" is used in the context of confidentiality.

At line 2, if $y = 1$ then x is marked sensitive because of control-dependent assignment in the then-clause. Since there is no assignment to z in the then-clause of line 2, it is not marked sensitive. Moreover, the condition at line 3 will not hold because x was assigned a value of 1 at line 2. But the condition at line 4 holds, so w is assigned the value of 1, but it is not marked sensitive since z is not sensitive at this point. Now, consider the case when $y = 0$. Following a similar line of reasoning, it can be seen that w will be assigned the value 0 at line 3, but it will not be marked sensitive. Thus, in both cases, w gets the same value as y, but it is not marked as sensitive.

As with control dependences, a malware writer can copy an arbitrarily large number of bits using nothing but implicit flow by simply using a slightly more sophisticated example of the above code. It is thus trivial for a malware writer to evade taint-tracking techniques that track only direct data dependencies and control dependencies.

2.2 Difficulty of Mitigating Evasion Attacks

To thwart control-dependence-based evasion, a taint-tracking technique can be enhanced to track control dependences. This is easy to do, even in binaries, by associating a *taint label* with the *program counter (PC)* [13][6]. Unfortunately, this will lead to an increase in false positives, i.e., many benign programs will be flagged as exhibiting remote-control behavior. To illustrate this, 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)) {
    fp = fopen("data", "w");
    ...
} else { ... // report error }
```

In the context of benign software, false positives due to control dependence tracking can be managed using developer annotations (so-called endorsement or declassification 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, especially when it is distributed in the form of binaries.

[6] Specifically, the PC is tainted within the body of a conditional if the condition involves tainted variables. Moreover, targets of assignments become tainted whenever the PC is tainted. Finally, the taint label of the PC is restored at the merge point following a conditional branch.

Mitigating implicit-flow based evasion is even harder. It has been shown that purely dynamic techniques cannot detect implicit flows [42]. This is because, as illustrated by the implicit flow example above, it is necessary to reason about assignments that take place on *unexecuted* program branches. On binaries, this amounts to identify the memory locations that may be updated on program branches that are not taken. Several features of untrusted COTS binaries combine to make this problem intractable:

- Address arithmetic involving values that are difficult to compute statically
- Indirect data references and indirect calls
- Lack of information about types of objects
- Absence of size information for stack-allocated and static objects (i.e., variables)
- Possibility that malicious code may violate low-level conventions and requirements regarding the use of stack, registers, control-flow, etc.

As a result, it is unlikely that implicit flows can be accurately tracked for the vast majority of today's untrusted software that gets distributed as x86 binaries.

2.3 Implications

Evasion measures described above can be mitigated by treating (a) all data written by untrusted code as tainted (i.e., not trustworthy), and (b) all data written by untrusted code as sensitive if any of the data it has read is sensitive. 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 over a coarse-grained, conservative technique that operates at the granularity of processes, and does not track any of the internal actions of a process.

In the context of detecting remote-control behavior, we observe that in the absence of evasion measures, the use of dynamic information flow techniques enables us to distinguish between malicious behavior, which involves the use of security-critical system call arguments that directly depend on untrusted data, and benign behavior. The use of evasion techniques can easily fool taint-tracking techniques that only reason about explicit flows. If the technique is enhanced to reason about control dependences, evasion resistance is improved, but as illustrated by the examples above, many more false positives are bound to be reported, thus significantly diminishing the ability of the technique to distinguish between malicious and benign behaviors. If we further enhance evasion resistance to address all implicit flows, we will have to treat all data used by an untrusted application to be tainted, thereby completely losing the ability to distinguish between benign and malicious behavior.

In summary, the emergence of practical dynamic taint-tracking techniques for binaries enabled high-precision exploit detection on trusted code. This was possible because the presence of explicit information flow from untrusted source to a security-critical sink indicated the ability of an attacker to exert a high degree of control over operations that have a high risk of compromising the target application — a level of control that was unlikely to be intended by the application developer. It seemed that a similar logic could be applied to untrusted code, i.e., a clear distinction could be made between acceptable uses of tainted data that are likely to be found in benign applications from

malicious uses found in malware. The discussion so far shows that this selectivity is lost once malware writers adapt to evade information flow techniques.

3 Analyzing Runtime Behavior of Shared-Memory Extensions

A significant fraction of today's malware is packaged as an extension to large softwares such as client-side web applications or the operating system. Applications such as web browsers and email clients are attractive targets for malware authors, because of the ubiquitous use of these applications in online financial transactions and private information exchange.

Nearly all large web browsers have software extension mechanisms that that allow adding various forms of additional functionality, such as better GUI services, automatic form filling, and viewing various forms of multimedia content. We refer to such browser extensions as browser helper objects (BHOs)[7]. Perhaps surprisingly, almost *all* browsers today have extensibility mechanisms that allow extension packages to be shipped with third-party libraries in binary form. Due to the growing user trends towards installing off-the-shelf extensions and due to increasing drive-by-downloads, malware spread in form of BHOs has been rampant.

Recent works [13] have proposed using information flow to track the flow of confidential data such as cookies, passwords and credentials in form-data as it gets processed by web browser. The idea is to monitor the actions of malware masquerading as benign BHOs, which is loaded in the address space of the browser, and to detect if confidential data is leaked by the BHOs. The crux of the problem is to selectively identify malware's actions. Essentially, their technique uses an attribution mechanism to classify actions that access system resources, to trusted and untrusted *contexts*. System calls or operations made directly by the BHO or by a host browser function called on its behalf, are attributed to the untrusted context, while those by the host browser itself belong to the trusted context. In the untrusted context, any sensitive data processed is flagged "suspicious." The presence of this data at output operations that perform writes to networks/files signals the leakage of confidential data effected 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 specific mechanisms proposed in these defenses. Below, we present several such evasion attacks. We remind our readers 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. In this section, our focus is on additional evasion techniques that become possible due to this shared address space.

3.1 Attacks using Arbitrary Memory Corruption

Corruption of untainted/insensitive data to effect leakage. By corrupting the memory used by the host application, a malicious BHO can induce the host application to carry out its tasks outside the untrusted context. For instance, a privacy-breaching malware *does not* necessarily need to read the confidential data itself and pass/copy it to

[7] Browser extensions are named in different ways. Internet Explorer uses the terms "BHOs", "extensions" and "toolbars", while Gecko-based browsers (e.g., FireFox) use the terms "plug-ins" and "extensions". We use the term BHO for all these terms interchangeably in the paper.

external network interfaces. Instead, it could corrupt the data used by the browser (i.e., the host application) such that the browser unknowingly leaks this information. We present the basic idea for an attack that avoids direct manipulation of any sensitive data or sensitive pointers. Instead, it corrupts higher level untainted pointers that point to the sensitive data. Consider a pointer variable p in the browser code that refers to data items to be transmitted over the network. A malware can corrupt p to point to sensitive data (say s) of its choice, stored within the browser memory. This way a malicious BHO can arrange for s to be transmitted over the network, without being detected by techniques described in [13]. Similarly, a BHO may corrupt a file descriptor 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 these attacks are rife in large systems. Moreover, they are easily forgeable because of the high degree of address space sharing between the host browser and extensions.

Optimistic assumptions about data originating from untrusted code. Another basic idea for attack involves using seemingly harmless data, such as constants, which are treated as untainted by most techniques [13,45] for corruption of browser data structures. Treating constants in untrusted code or any data under the control of the malware as untainted is anyway problematic, and specially so in binary code where constants may be addresses. The attack involves overwriting an untainted pointer p, that may initially point to a sensitive data s, with an untainted value such as constant memory address m. When the browser uses m for a critical operation, such as determining the destination for sending s, this threat becomes very significant as shown below.

A real attack. We now present an example that illustrates how a BHO can corrupt a data pointer to violate a policy that prevents leakage or tampering of sensitive information, like the user's *cookies*, by the BHO. The example has been tested on Lynx, a textual browser which does not have a proper plugin framework support[8]. However, it uses libraries to enhance its functionalities and, as they are loaded into Lynx's address space, they can be considered as untrusted components. In fact, the attack's result could be applied to a different browser application (e.g., Internet Explorer, FireFox) with a full-blown plug-in framework.

The attack consists of modifying the domain name in the cookie, and is illustrated in the figure below. 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). Subsequently, when the browser has to send a cookie, the domain is compared using `host_compare` (not shown) which calls `stringcasecmp`. A plug-in can traverse the linked list, and write its intended URL to the `domain` pointer field in cookie record. On enticing the user to visit a malicious web site, such as evil.com, these cookies would automatically be sent to the attacker web site, thereby subverting the implementation of the Same Origin Policy. The point to note in this example is that the `domain` pointer will be untainted; the object it points to will be tainted or sensitive. These higher level pointers themselves are not sensitive, therefore they can be corrupted without raising suspicion.

[8] Lynx has been chosen to simplify the example.

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

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

Implications

The above example shows how confidential data can leak without being read. The approach proposed in [13] does not deal with this threat. Recall that sensitive data is marked "suspicious"(to use the terminology defined in [13]), only when the untrusted BHO uses the sensitive data itself or propogates it to the external interfaces. Consequently, the malware can overwrite the `domain` pointer with an address value (which is untainted) of choice, *without* causing the *suspicious flag* to be set.

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[9]. Considering every write performed by the untrusted BHO to be tainted, as suggested previously (therefore, considering everything written by the untrusted BHO as "suspicious"), may be a too conservative a strategy. It may yield high false positives in the cases where plugins access sensitive data but do not leak it. Though, applying conservative tainting specifically to recognize control data as done in [44] seems reasonable, this may raise significant false positives when applied for identifying all data that is possibly controlled by the plugin.

3.2 Attacking Mechanisms Used to Determine Execution Context

In a shared memory setting, it is necessary to distinguish the execution of untrusted extension code from that of trusted host application code. To make this distinction, the detection approach needs to keep track of a code execution *context*. The logic used for

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

maintaining this context is an obvious target for evasion attacks: if this logic can be confused, untrusted code could execute with the privileges of trusted code. A more subtle attack involves data exchanged between the two contexts. Since execution in trusted context affords more privileges, untrusted code could achieve its objectives indirectly by corrupting data (e.g., contents of registers and the stack) that is 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 evasion attacks will need to rely on these details. Below, we describe how such evasion attacks can work in the specific context of [13].

Attacking context-switch logic. The approach proposed [13] for context tracking uses the following algorithm. For each instruction, the system checks whether the instruction belongs to the BHO code region. If so, then it saves the value of the current stack pointer as esp_{saved}, and the instruction is executed in untrusted context. Whenever the instruction pointer points outside the code region of the BHO, the system has to determine whether the instruction is executed on behalf of the BHO (i.e., untrusted context) or not. For this, the proposed technique utilizes the fact that on their platform the stack grows downwards and checks if the current stack pointer,$esp_{current}$, is below the esp_{saved}. The context identification logic implicitly assumes a benign call stack model – it assumes that the activation records are pushed on the stack, the stack data belonging to the caller is left unchanged by the callee, and that the callee function cleans up its activation leaving the stack pointer restored after its invocation. We point out that these assumptions are reasonable for calls across benign code modules only. Specifically, if the $esp_{current}$ is not less than esp_{saved}, the context switching logic assumes that the last untrusted BHO code stack frame has been popped off the activation stack and the execution context does not belong to the BHO anymore. This attribution mechanism allows valid (benign) context switches (from untrusted to trusted context) at call/return function boundaries, when the last BHO function f is about to return and there are no other browser functions invoked by f.

Unfortunately, we show that this attribution mechanism is insecure. Malware may employ simple low-level attacks that subvert the control flow integrity of the application at the host-extension interface leading to devastating attacks. The taint analysis approach and the attribution mechanism employed in [13] point out that the mechanism can deal with two threats that may circumvent context attribution – execution of injected code, and attempts to adjust the stack pointer above the threshold limit by changing the ESP register in its code. However, it does not protect against other low-level integrity violations, such as return-into-lib(c) style [31,35] attacks, which aim to eventually execute already present code.

To be concrete, consider the scenario where the malicious BHO corrupts control pointers, such as return addresses pushed by the calling host function, to refer to target locations in the browser or its trusted libraries. It could additionally create a compatible stack layout required for a return-into-lib(c) attack to perform intended action and let its last invoked function simply exit. Changing control pointers such as return address above the recorded threshold stack pointer value, without making any modification to ESP itself, is sufficient and touches no sensitive/tainted data. Such returns from un-

trusted code trigger control transfers to the attacker controlled target functions, and furthermore, with arbitrarily controlled parameters on the crafted stack layout. As no other BHO instructions are executed after such a return, subsequent code will be executed in the browser context fulfilling the attacker's objectives.

Implications

To counteract such a return-into-lib(c) style attack, a malware analysis has to strengthen the attribution mechanism, to allow information flow to be correctly captured for the different contexts.

Another work in this area, Panorama [45], proposes to label every write operation performed by a BHO for the purpose of being able to track dynamically generated code. But, it seems to rely on a similar attribution mechanism used in [13], and seems vulnerable to the attack presented in the previous section as the attribution mechanism can be circumvented. HookFinder [44], instead, is able to catch every hook implanted into the system by an untrusted binary. To do so, they use an approach which is similar to information flow-based techniques: they label every write operation performed by untrusted binaries, as they want to be able to analyze any hooking attempts (regardless it they are made by benign or potentially malicious modules). This seems to be a promising approach for the attribution problem. In fact, an extension to their strategy, as the one proposed in [33], which marks context as untrusted whenever control transfers involve tainted pointers resolves the issue of correctly attributing context.

3.3 Attacking Meta-Data Integrity

Corrupting meta-data maintained by a dynamic information flow technique is another avenue for attack. Typically, meta-data consists of one or more bits of taint per word of memory, with the entire metadata residing in a memory-resident data structure 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 [13] can store metadata in the emulator's memory, which cannot be accessed by the emulated program. Other techniques such as [43] 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.

Attacks based on data/meta-data races. Dynamic information flow techniques need to 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:

$time$	**Benign Thread**	**Malicious Thread**
t_1		set tag_x to *tainted*
t_2	set tag_x to *untainted*	
t_3	write *untainted* value to x	
t_k		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 10 (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  void check_preferences () {
3      ...
4      if (get_pref_name () == OK)
5          old_fname = "/.../.mozilla/.../pref.js";
6      ...
7      while (...) {
8          fname = old_fname;
9          if (fname) {
10             fp = open (fname, ''w'');
11         ...
12     }
13 }
```

(a)

```
1  void *malicious_thread(void *q) {
2      int attempts = 0;
3      while (attempts++ < MAX_ATTEMPTS)
4          fname = "/.../.mozilla/.../cookies.txt";
5
6
7
8
9
10
11
12
13 }
```

(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 [43], with all taint tracking code inlined, as this minimizes the number of instructions for taint tracking and hence the vulnerability window. Assuming that the `get_pref_name` call fails to return `OK`, 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. The reason why this happens is because the transformed benign thread reads the taint for `fname` on line 8 and sets the control context to tainted scope, before executing the original code for performing conditional comparison on line 9. The malicious thread tries to interleave its execution with the one of the benign thread, trying to achieve the following ordering of operations on the shared variable `fname`:

Time	Operation	Thread (Line No.)
t_1	read $tag_{fname} \mapsto$ *untainted*	Benign (9)
t_2	write $tag_{fname} :=$ *tainted*	Malicious (4)
t_3	write $fname :=$ "/home/user/.mozilla/.../cookies.txt"	Malicious (4)
t_4	read $fname$	Benign (9)

If such an ordering occurs, the tag_{fname} read by the benign thread is marked *untainted* as the benign thread has cleared the taint previously, while the data happens to contain an attacker controlled value about user browser cookies. Consequently, contrary to the intention of the instrumentation of tracking control-dependence, the attacker manages to prevent control scope from switching to tainted scope at line 9 in the benign code. In practical settings, the window of time between t_1 and t_4 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.

Implications

Attacks on direct corruption of metadata has been studied before [43] and thwarted by implementations using virtual machines and emulators which explicitly manage the context switches between threads or processors. However, much of the design of such metadata tracking monitors has not been carefully studied in the context of multi-threaded implementations (or multi-processor emulators), and techniques in this section highlight the subtle importance of these.

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.

In order to expose such trigger-based behavior, Moser *et al.* [1] suggested an interesting dynamic technique that combines the benefits of a static and dynamic information-flow analyses. Specifically, they taint trigger-related inputs, such as calls to obtain time, or network reads. Then, dynamic taint-tracking 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 a snapshot of the analyzed process, and keeps exploring one of the branch. Subsequently, when the exploration of the taken branch ends or after a timeout threshold is reached, their technique forces the execution of the unexplored branch. Such forcing requires changing the value of a tainted variable v used in the conditional, so that the value of the condition expression is now negated. By leveraging on a *decision procedure* to generate a suitable value for v, the proposed approach also identifies any other variables in the program whose values are dependent on v, and modifies them so that the program is in a consistent state[10]. 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, or registry entry. The approach proposed 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.

Since the underlying problems the analysis proposed by Moser *et al.* has to face are undecidable in general, their technique is incomplete, but seems to work well in practice against contemporary malware. However, this incompleteness can be exploited by a malware writer to evade detection. For instance, 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, the approach proposed in [1] 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. Nonetheless, today's malware writer places high value on stealth, and hence would prefer alternative anti-analysis mechanisms that do not raise suspicions, and we describe such primitives next.

4.1 Evasion using Memory Errors

Binary code is generally hard to analyze, as briefly pointed out in Section 2.2. For instance, this is due to the absence of information about variables boundaries and types, which makes many source-based analyses inapplicable to binaries. We observe that given an arbitrary binary, it is hard to say whether it potentially contains a vulnerability such as a memory error (e.g., buffer overflow), and to determine the precise inputs to

[10] 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.

exploit it. Exhaustively running the binary on all possible inputs is often infeasible for benign code, leave alone malware which is expected to exploit the exponential nature of exhaustive searches to cause the worst-case hit each run.

Motivated by this observation, we present an attack against dynamic information flow-based analyses used to analyze malware behavior, similar to the one presented in [1]. This attack is able to hide malicious code from being discovered and further strengthen it such that extensions to analysis employed in [1] are unable to detect it. Our attack leverages on the introduction of *memory errors*, as shown in the following example.

```
1  int trigger;
2  ...
3  void procInput(void) {
4      int *p = &buf[0];
5      char buf[4096];
6      ...
7      my_gets(buf);
8      ...
9      *p = 1;
10     ...
11     if (trigger)
12         malcode();
13 }
```

The introduced memory error is a plain stack-based buffer overflow vulnerability[11]. The attacker's goal is to write past the end of `buf` (line 7) and corrupt the pointer `p` to make it point to the variable `trigger`. Eventually, when the vulnerability will be exploited, the malware will set `trigger` to 1 (line 9) which in turn has the effect to disclose the malicious code represented by `malcode()` at line 12, guarded by `trigger`. It can be observed that the lack of proper bound checking in the code snipped shown above is not to be considered as a suspicious pattern by itself. The mere use of an unsafe function as `my_gets`[12] does not imply that there is a memory error. In fact, bound checking could have been performed elsewhere by the programmer (which justifies the use of an unsafe function), or the programmer knows that at that point the input can never be bigger than `buf`.

In order to disclose the malicious code during analysis, the variable `trigger` has to eventually be marked as tainted, so that the code it guards can be further analyzed. The variable `trigger` is never tainted unless `p`, which can potentially be corrupted with tainted data by the malware, points to it. The problem of determining whether `p` could point to `trigger` is undecidable statically, thus augmentations to [1] using some form of static analysis do not help. On the other end, one might argue that the dynamic approach proposed in [1] could potentially accomplish the *detection* of the overflow, at least (while it is unlikely that the correct vulnerability exploitation can be achieved). In fact, given the aforementioned example, it is fairly easy for the analysis technique considered to generate a big-enough input which will eventually corrupt the pointer `p`.

[11] It is important to note that there are no constraints on the type of vulnerability introduced. A generic buffer overflow, an integer overflow, or a (custom) format string vulnerability would have done as well.

[12] This function resembles the well-known libc `gets`. The malware author can either use its own implementation or the one provided by the C library.

Even if such a technique is employed, we show that we can extend this example to make it even harder – if not unfeasible – to achieve this step.

To this end, it would be desirable to have a function f that is easy to compute, but hard to reason about some properties of it. By doing so, it is possible to modify the previous example in such a way to make it harder for the analyzer to even detect whether a memory error vulnerability is present or not. Such a situation is depicted by the following code snippet (the action performed by this code can be found in benign program as well).

```
...
int trigger;
...
void procInput(void) {
    int pad, n, l;
    char buf[4096+256];
    int *p = &pad;
    char *dst;

    ...
    n = read(s, buf, sizeof (buf));
    l = computespace(buf, n);
    // make sure we have enough room
    dst = alloca(l + 128);
    decode(buf, l, dst);
    ...
    *p = 1;
    ...
    if (trigger)
        malcode();
    ...

}
```

```
int computespace(char *src, int nread) {
  int i, k = 0;
  for (i = 0; i < nread; i++) {
    switch(src[i]) {
      case 0: k++; break;
      ...
      case 255: k++; break;
    }
  }
  return k;
}

void decode(char *src, int nread, char *dst) {
  int i, j;
  for (i = 0, j = 0; i < nread; i++, j++) {
    switch(src[i]) {
      case 0: dst[j] = src[i]; break;
      ...
      case 113: dst[j++] = src[i];
                dst[j] = src[i];
                break;
      case 114: dst[j] = src[i]; break;
      ...
      case 255: dst[j] = src[i]; break;
    }
  }
}
```

It is worth noting that the function `computespace` is easy to compute, but is relatively hard to reason about some properties of it. For instance, by looking at the source code, it is easy to understand that at the end of the computation `k` holds the same value as the length of the data read into the buffer `buf`. On the other end, the same reasoning can be hard to do on binaries and in an automated way. Thus, it is hard to correlate `n`, the number of read bytes, to `l`, the minimum number of space to allocate to be sure the function `decode` does not cause overflow. The function `decode` presents a problem by itself, by deliberately introducing the condition for an overflow to occur. In fact, it can cause `dst` to overflow into `p` if the number of bytes given as input (`buf`) whose ASCII value is 113 exceed a certain threshold. Only an exhaustive search over all the possible input values and combination would deterministically trigger this memory error. Unfortunately, such an enumeration would be extremely onerous if not impossible to perform. Similar to NP-complete problems which are hard to solve while verification of correct answers is easy, it is rather simple for the attacker to provide the right input which will cause `dst` to overflow so that `p` can be corrupted in such a way to eventually disclose the malicious behavior. From the analysis point of view, instead, an exhaustive search will probably start with a sequence of length 1, trying all the possible 256 ASCII

values. This does not cause overflow as there is a safe padding of 128 bytes for `dst`. Following this reasoning, a sequence of length k and 256^k combination have to be tried. For instance, a k equal to 128 can reach the boundaries of `dst` (`dst` length is 256 bytes and 128 times 'q' reaches its boundaries). This, however, would roughly require to test 255^{127} combinations to try out on average which is a fairly huge number.

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 `malcode()` 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.

4.2 Implications

The implications on whether dynamic information flow-based techniques can help to disclose, analyze, and understand the behavior of the next-generation of malware is similar to the ones pointed out in the rest of this paper. In fact, to detect the evasion technique proposed in the previous section, an information flow-based approach should ideally be able to trigger *any* memory error which may be present in the analyzed software, and automatically exploit the vulnerability so that interesting (i.e., tainted) previously disabled conditions will be examined. In the previous section we have shown how this could be hard – if not impossible – at all to achieve, if directly faced. Alternatively, information flow analyses could taint *any* memory location, considering all the possible combinations, and see how information is propagated. While this would eventually taint `trigger` and thus disclose the malicious behavior, it would drop the benefits provided by taint-tracking mechanisms which focus the analysis on *interesting* data, as *every* paths would be forced to be explored. For instance, the resulting analysis would be similar to the one proposed in [9] where, even if the underlying technique is different, the end result is that *every* path can potentially be explored, which of course is a hard task by itself. For instance, one may attempt to force execution of all program paths, but this is likely to be very expensive, and to suffer from semantic inconsistencies that may arise due to forcing execution down branches that are not taken during execution.

5 Related Work

Information flow analysis has been researched for a long time [6,12,14,20,23,32,41]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [6]. More recent work has been focused on language-based approaches, capable of tracking information flow at variable level [27]. 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. Moreover, they typically work with simple, high-level languages, while much of security-critical contemporary software is written in low-level languages like C that use pointers, pointer arithmetic, and so on. Finally, it can be noted that despite their benefits static analyses are generally vulnerable to obfuscation scheme, as recently remarked by [22]. Therefore, it is reasonable to rely on dynamic or hybrid approaches, instead. As a result, information flow tracking for such software has been primarily based on run-time tracking of explicit flows that take place via assignments.

Recently, several different information flow-based approaches have been proposed in the literature [11,15,16,30,36,43]. They give good and promising results when employed to protect benign software from memory errors and other types of attacks, 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 to embed evasion attacks, such as covert channels, and general memory corruption.

Probably, an ideal solution would require that untrusted binaries would carry proofs that some properties are guaranteed. This is achieved by proof-carrying code [25]. To be successful, this technique relies on some form of collaboration between the code producer and consumer. For instance, Medel *et al.* [21] and Yu *et al.* [46] proposed information flow analyses for typed assembly languages. Likewise, Barthe *et al.* provided non-interference properties for a JVM-like language [4] and dealt with timing attacks by using ACID transactions [5], as well. Unfortunately, it is unlikely that malware writers (i.e., the code producer, in this context) are going to give this form of collaboration which is necessary for the success of these approaches. Therefore, it is unlikely that these strategies would soon be adopted as is in the context of malicious software analysis and containment.

Driven by the recent practical success of information flow-based techniques, several researchers have started to propose solutions based on dynamic taint analysis to deal with malicious or, more generally, untrusted code [1,13,29,34,40,44,45]. During the last years, these techniques 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, information flow is a fragile technique that has to be supported by new analyses to be more resilient to evasions purposely adopted by ever-evolving malware.

6 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. Kruegel and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 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. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5), 2000.
4. G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. *Programming Languages and Systems*, 2007.
5. G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. In *In Proceedings of 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.
6. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., 1973.
7. F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, 2005.
8. K. J. Biba. Integrity considerations for secure computer systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.
9. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, 2006.
10. S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.
11. S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. 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)*, 2005.
12. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.
13. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Usenix Tech Conference*, 2007.
14. J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2), 1974.
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. J. Kong, C. C. Zou, and H. 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*, 2006.
17. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6), 2005.
18. McAfee. W32/hiv. virus information library, 2000.
19. McAfee. W32/mydoom@mm. virus information library, 2004.
20. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, 1994.
21. R. Medel. *Typed Assembly Languages for Software Security*. PhD thesis, Department of Computer Science, Stevens Institute of Technology, 2006.
22. A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. *ACSAC*, 2007.
23. A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM POPL*, pages 228–241, 1999.
24. S. Nanda, W. Li, L. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *IEEE/ACM Conference on Code Generation and Optimization (CGO)*, 2006.
25. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, 1997.
26. N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, 2007.
27. Perl. Perl taint mode. `http://www.perl.org`.
28. T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.
29. G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4), 2006.
30. F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *IEEE/ACM International Symposium on Microarchitecture*, 2006.
31. Rafal "Nergal" Wojtczuk. The Advanced return-into-lib(c) Exploits: PaX Case Study. Phrack Magazine, Volume $0x0b$, Issue $0x3a$, Phile $\#0x04$ of $0x0e$, 2001.
32. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), 2003.
33. P. Saxena, R. Sekar, and V. Puranik. A practical technique for integrity protection from untrusted plug-ins. Technical Report SECLAB08-01, Stony Brook University, 2008.
34. 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)*, 2007.
35. Clad "RORIV" Strife and Xdream "ROJIV" Blue. Ret onto Ret into Vsyscalls.
36. 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*, 2004.
37. P. Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.
38. TrendMicro. Bkdr.surila.g (w32/ratos). virus encyclopedia, 2004.
39. A. Vasudevan. *WiLDCAT: An Integrated Stealth Environment for Dynamic Malware Analysis*. PhD thesis, The University of Texas at Arlington, USA, 2007.
40. 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)*, 2007.
41. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3), 1996.
42. D. M. Volpano. Safety versus secrecy. In *SAS*, 1999.
43. W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.
44. H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *NDSS*, 2008.
45. H. Yin, D. Song, E. Manuel, C. Kruegel, and E. 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)*, 2007.
46. D. Yu and N. Islam. A typed assembly language for confidentiality. In *ESOP*, 2006.