7-1-2011

Fine-grained reasoning about the security and usability trade-off in modern security tools

Mohammed I. Al-Saleh

Mohammed Al-Saleh

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

, Chairperson

Fine-grained Reasoning about the Security and Usability Trade-off in Modern Security Tools

by

Mohammed I. Al-Saleh

B.S., Jordan University of Science and Technology, 2003
M.S., New Mexico State University, 2007

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2011

Dedication

To my parents for love, strong emotions, and care.

To my wife Haifa for love, endless support, and understanding.

To my daughters Leena and Jana.

To my brother, sisters, and friends who cared about me for a long time.

Acknowledgments

I would like to thank my advisor, Jedidiah Crandall, for his continuous support and encouragement in all aspects at all times. Also, I would like to thank my dissertation committee, Dorian Arnold, Terran Lane, and Rafael Fierro for their time and efforts.

Fine-grained Reasoning about the Security and Usability Trade-off in Modern Security Tools

by

Mohammed I. Al-Saleh

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2011

Fine-grained Reasoning about the Security and Usability Trade-off in Modern Security Tools

by

Mohammed I. Al-Saleh

B.S., Jordan University of Science and Technology, 2003
M.S., New Mexico State University, 2007

Ph.D., Computer Science, University of New Mexico, 2011

Abstract

Defense techniques detect or prevent attacks based on their ability to model the attacks. A balance between security and usability should always be established in any kind of defense technique. Attacks that exploit the weak points in security tools are very powerful and thus can go undetected. One source of those weak points in security tools comes when security is compromised for usability reasons, where if a security tool completely secures a system against attacks the whole system will not be usable because of the large false alarms or the very restricted policies it will create, or if the security tool decides not to secure a system against certain attacks, those attacks will simply and easily succeed.

The key contribution of this dissertation is that it digs deeply into modern security tools and reasons about the inherent security and usability trade-offs based on identifying the

low-level, contributing factors to known issues. This is accomplished by implementing full systems and then testing those systems in realistic scenarios. The thesis that this dissertation tests is that we can reason about security and usability trade-offs in fine-grained ways by building and testing full systems. Furthermore, this dissertation provides practical solutions and suggestions to reach a good balance between security and usability. We study two modern security tools, Dynamic Information Flow Tracking (DIFT) and Antivirus (AV) software, for their importance and wide usage.

DIFT is a powerful technique that is used in various aspects of security systems. It works by tagging certain inputs and propagating the tags along with the inputs in the target system. However, current DIFT systems do not track implicit information flow because if all DIFT propagation rules are directly applied in a conservative way, the target system will be full of tagged data (a problem called overtagging) and thus useless because the tags tell us very little about the actual information flow of the system. So, current DIFT systems drop some security for usability. In this dissertation, we reason about the sources of the overtagging problem and provide practical ways to deal with it, while previous approaches have focused on abstract descriptions of the main causes of the problem based on limited experiments.

The second security tool we consider in this dissertation is antivirus (AV) software. AV is a very important tool that protects systems against worms and viruses by scanning data against a database of signatures. Despite its importance and wide usage, AV has received little attention from the security research community. In this dissertation, we examine the AV internals and reason about the possibility of creating timing channel attacks against AV software. The attacker could infer information about the AV based only on the scanning time the AV spends to scan benign inputs. The other aspect of AV this dissertation explores is the low-level AV performance impact on systems. Even though the performance overhead of AV is a well known issue, the exact reasons behind this overhead are not well-studied. In this dissertation, we design a methodology that utilizes Event Tracing for

Windows technology (ETW), a technology that accounts for all OS events, to reason about AV performance impact from the OS point of view. We show that the main performance impact of the AV on a task is the longer waiting time the task spends waiting on events.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The world is now more connected than ever because of the Internet. Internet users need security as a basic connection requirement because nefarious hackers have strong incentives to break into users' machines for different purposes. Users use off-the-shelf or open-source security tools to secure their machines. Those security tools should achieve two conditions: First, they should secure the target system from attacks; Second, they should not disturb the target system's normal functionality. In other words, the target system should be usable[1], as it was before adding the security tool. However, the balance between these two conditions is always a challenge to achieve. For example, a system that preserves confidentiality through very strong encryption algorithms might not be practical to use. Security systems should always be designed towards achieving a good balance between security and usability.

Saltzer and Schroeder [83] present eight security design and implementation principles for system protection. One of those principles is "*Open design*: the design should

[1]Usability in this dissertation means the ability to use the target system after deploying a security tool in a normal way. If the security tool kills the performance or shortens the lifetime of the target system then it is not usable according to our definition. Interface design principles for security tools to make them easy to use is out of the scope of this dissertation.

not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords." So, security tools should always be examined by researchers or developers to look for deficiencies. This process leads to better, more secure systems. The goal of this dissertation is to examine modern security tools aiming at improving their security and usability tradeoffs. We study two modern security tools, Dynamic Information Flow Tracking (DIFT) and Antivirus (AV) software, for their importance and wide usage. We precisely reason about the main factors that cause known issues in both tools by studying their internals and experimenting with their functionalities. This is achieved by implementing full systems and testing them in realistic scenarios.

Dynamic Information Flow Tracking is a multi-purpose and powerful technique. Its usage includes attack detection [34, 36, 80], analyzing data lifetime in a full system [28], malware analysis [103, 102, 18], network protocol analysis [62, 99], and dataflow tomography [69]. DIFT works by tagging data and following it throughout the system. Whenever the tagged data is involved in an operation of interest, a measure is taken. Suh *et al.* [91] divided information flow into five types of dependencies that DIFT systems must track to be complete: copy dependencies, computation dependencies, load-address dependencies, store-address dependencies, and control dependencies. However, current research does not track the last three in the general case, because if those dependencies are completely applied the whole system will be tagged (overtagging) and thus the DIFT system reveals very little about the actual flow of information.

In this dissertation, we define a tag to be the mutual information between a source and a destination. Thus, a tag is a quantity of mutual information and is no longer binary (0 or 1). In our scheme, we reason about fine-grained contributing reasons to the overtagging problem. When one source of information is copied into many destinations (for example through a loop) and then some of those destinations are combined together later in another operation, the tags of those destinations should not added, mainly because

adding them would increase the tag value even more than its original value from the original source. Since a tag represents mutual information with a taint source, if two tags have mutual information amongst themselves then their information is not independent. Also, in certain situations, a tag value should be decreased (or even not transferred) based on the corresponding operation. For example, *xor*ing a register with itself should wipe out its tag value, because the information is completely destroyed. Another example is when a tagged address is used to zero out a series of memory locations (like an array) then the address tag should not be copied into the memory locations' tags. This suggests that checking the sources of tags and the context of operations are the main things to consider to decide how much information is being transferred and consequently how much mutual information is being propagated. We also claim that the complication of devices and operating systems contributes to the problem of overtagging. In this dissertation, we propose a practical way to deal with all of these aspects. We have applied DIFT as an intrusion detection system in two different target systems: sensor network devices and general purpose computers.

Sensor network devices are widely used these days in different venues, including military and medical applications. Providing security for these devices is required due to the importance of the information they deal with. Sensor devices have design constraints that are different from those in general purpose computers. Any security mechanism for sensor devices should not degrade the performance. If the right information is delivered at the wrong time, then that could cause a disaster. Also, security mechanisms should be economical, in terms of consuming the device battery, because it is almost impossible to change the battery in hostile environments. Furthermore, although patching sensor application software after deployment is technically doable [60], it is often not possible. So, it is required to have a completely secure and usable system before deployment. In Chapter 2, we provide the required system to secure sensor network devices against remote control attacks. We show that DIFT, in its regular binary tag implementation, can be used to secure sensor network devices in an efficient way because of the simplicity of the sensor devices and operating systems.

Chapter 3 presents a novel approach to apply all DIFT propagation rules in general purpose computers, as an intrusion detection system, while maintaining both security and usability. In our system, tags are mutual information between data sources and destinations. Also, we introduce the idea of tag lifetime by throttling tags based on their usage to reduce the overtagging problem. As a motivating example to demonstrate this system, we define an attack by the amount of control that external network entities have over what a networked system is doing. We measure the amount of information flowing between tainted sources and the control path of the CPU for a variety of scenarios and show that our prototype system gives intuitive, meaningful results.

The second security tool we test, besides DIFT, is the Antivirus (AV) software. The AV cannot detect all attacks, but is very effective against known threats and is a first line of defense against some types of novel threats. A Windows system keeps notifying a user if AV is not installed or is removed from the system. Although we agree that the AV is an important security tool, it is susceptible to evasion, especially through new attacks which the AV has no signatures for. Because there is little effort to study AV security and usability, given how important it is, we do so in this dissertation. In Chapter 4, we examine the AV internals looking for a possibility of timing channel attacks against the AV based on how the AV scans data against its database of signatures, aiming to enhance the AV security if such a possibility exists. The purpose of this attack is to see if the AV is updated with certain signatures or not, only based on time. An attacker could use this information to know how up-to-date a user's AV is without triggering any alerts. Even though the AV makes the common cases (scanning purely benign data) fast to achieve better performance (usability), it leaks information when it deals with non-common cases; so the attacker can infer if certain inputs mean something to the AV or not by measuring the scanning time of carefully crafted inputs.

As a complementary study, in Chapter 5, we test the AV impact on the performance of common tasks from the operating system point of view. In this study, we want to show

how intrusive the AV is to those tasks. The less intrusive and the better performance the AV, the more usable. To get accurate and meaningful results, we use the Event Tracing for Windows technology to account for all OS events down to context switches while running common tasks. Then by dumping the events' information into an SQL database, we are able to design queries to aggregate information about what happened, and by which process or thread, over time. We show that the main performance impact of the AV on a task is the longer waiting time the task spends waiting on events.

Chapter 2

Secure and Usable DIFT System for Sensor Network Devices

Sensor network devices are no less vulnerable to remote attacks, such as malicious worms, than their general purpose computer counterparts, and are presented with unique threats because of the hostile environments sensors are placed in. It is well known that sensor devices place challenging constraints on any attempt to secure them against these attacks, including small performance and power budgets, infrequent patch updates, and long service lives. However, in this dissertation we demonstrate that security *can* be built into sensor devices "from the ground up."

In this dissertation we apply dynamic information flow tracking (DIFT) to sensor devices, where network data is tagged as untrusted and then these tags propagate throughout the system. Our results demonstrate that minor hardware modifications to sensor devices can provide sufficient security guarantees against remote control data attacks. To make these guarantees we address all five dynamic information flow dependency types (copy, computation, load-address, store-address, and control), whereas DIFT schemes for general purpose computers are empirically only able to address the first two. Rigorous testing

of eight applications shows that no modifications to existing operating systems, compilers, applications, or binaries is necessary.

2.1 Introduction

Securing sensor networks before they are deployed is critical. Today, these systems are being deployed in military and medical applications, yet are still built on top of architectures with simple memory models using the C language, or variants of C such as nesC [48]. Memory corruption vulnerabilities can lead to worm attacks [50] and other remote intrusions that allow adversaries to completely take control of the network. Even Harvard architectures, where instructions and data are kept in separate memories, are susceptible to these kinds of attacks [46].

At the same time, compared to general purpose computers, sensor devices have design constraints that place even more restrictions on security mechanisms designed to thwart these attacks. In addition to constrained performance and power budgets, sensors are often placed in hostile or remote environments. Thus, while patching sensor application software after deployment is technically feasible [60], it is often not possible. Building security into the devices "from the ground up" through a secure architecture, without requiring modifications to the application source code, compiler, operating system, or program binaries, is imperative.

The *dynamic information flow tracking* (DIFT) scheme that we present in this chapter marks all data that is read from the network (typically from a radio device) as untrusted using a *tag bit*, sometimes also called a *taint bit* (throughout this chapter we use the terms untrusted and tainted interchangeably). Tags are propagated throughout the system by the architecture, with no need for modification to the program binaries. No untrusted data can be used as the target address for a control flow transfer such as a jump, call, or return. This

makes attacks that overwrite *control data*, such as buffer overflows and related attacks, impossible. Control data includes return addresses, function pointers, the bases and offsets of linked library functions, and more. Thus the architecture we present stops any attack that overwrites control data and hijacks control flow to take control of a network sensor, and we discuss in Section 2.5 how non-control data attacks and attacks at higher levels of abstraction could also be addressed with this scheme.

The key insight behind our work is that sensor network applications have very regular and predictable patterns of memory management, when compared to general purpose systems. DIFT [91, 34, 36, 73, 32, 80] has been proposed as an attractive solution to memory corruption vulnerabilities because, if all possible information flow dependencies are addressed, DIFT schemes can secure commodity code with very minor modifications to the hardware. Unfortunately, memory management in general purpose systems is very complex, so that in DIFT schemes for these systems the load- and store-address dependencies, where information can flow based on the address with which a piece of data is loaded or stored, cannot be tainted without rendering the system unusable due to false positives [89]. By not tracking load- and store-address dependencies, current DIFT schemes are open to attacks that use multiple levels of pointer indirection [33, 38, 78]. One study [36] even found that existing attacks not designed to subvert DIFT can evade detection and reduce existing DIFT schemes to the same level of security as NX (non-executable) pages, which has been shown to not stop attacks even when combined with various other limitations on the attacker [86, 24].

What we demonstrate in this chapter, however, is that DIFT schemes for sensor networks need not be as limited, and can provide real security guarantees. We have implemented a DIFT scheme on the MICA2 platform that is secure against all control data attacks. Through rigorous testing of eight applications, we demonstrate that tracking load-/store address dependencies does not break existing applications nor require any modifications to the application source code, program binary, compiler, or operating system.

Then through dynamic testing and manual code inspection of these eight applications we determine that all control dependencies in the applications are benign. Thus, whereas existing DIFT schemes have **not** provided satisfactory security guarantees against remote control flow hijacking attacks on general purpose systems, our scheme demonstrates that it is possible to do so for sensor network applications.

Specifically, our contributions are the following:

1. We present a DIFT scheme for sensor network nodes, based on the MICA2 platform, that provides satisfactory security guarantees against remote control data attacks without requiring any modifications to existing application source code, program binaries, compilers, or operating systems and only minor modifications to the hardware.
2. We demonstrate the security and usability of this scheme by testing eight applications dynamically and through manual inspection.
3. We elaborate on Suh *et al.*'s classification of information flow dependency into five types (copy, computation, load-address, store-address, and control) to more clearly define what it means for a DIFT scheme to be secure.
4. We discuss how securing sensor nodes from remote control flow hijacking based on control data corruption can serve as a basis for security against other remote attacks, such as non-control data attacks [27].

2.2 Definitions

In this section we discuss how information can flow, using Suh *et al.*'s [91] categorization of information flow dependencies into five types: copy dependency, computation dependency, load-address dependency, store-address dependency, and control dependency. This section is meant to make it clear what is and is not secure with regards to how a particular

```
for (Loop = 0; Loop < Size; Loop++)
     ConvertedArray[Loop] = LookupTable[InputArray[Loop]];
```

Figure 2.1: **An example of a load-address dependency.**

DIFT scheme tracks these dependencies.

2.2.1 Terms

A *remote control flow hijacking attack* is when an attacker sends data to a remote computer or sensor and causes the control flow of a service to be diverted to the code of the attacker. Without user intervention, virtually all remote attacks such as worms are typically of this form. Also note that the code of the attacker need not be machine instructions, return-into-libc [70] or more advanced attacks [86, 24] are possible. Also, control flow can be hijacked at higher levels of abstraction, such as a perl interpreter.

Control data attacks are the most common form of remote control flow hijacking attacks, in which control data such as return addresses, function pointers, and jump targets are overwritten by the attacker so that low-level control flow is diverted. These attacks exploit memory corruption vulnerabilities such as buffer overflows [75], double `free()s` [14], format string vulnerabilities [85], and more. *Non-control data attacks* exploit the same kinds of memory corruption vulnerabilities to overwrite types of data other than control data, therefore taking control of the system without directly corrupting control data.

Entropy is a measure of uncertainty regarding the occurrence of an event.

2.2.2 Copy and computation dependencies

A copy dependency occurs when data is copied from one storage object to another, such as register-to-register, memory-to-register, or register-to-memory. When this happens, the

tag bit of the destination should be set to the value of the tag of the source. A computation dependency occurs when an operation such as an arithmetic operation or bit manipulation is performed on one or more source registers, and the result is written to a destination register. In this case, most DIFT schemes will apply a low-water-mark policy, where the destination is tainted as untrusted if *any* of the source registers are tainted. Sometimes an exception is made for the `XOR` operation since `XOR`ing a register with itself is a common way to zero out the register [91]. We did not find this to be necessary on the MICA2 platform. Most DIFT schemes for general purpose computers are able to handle copy and computation dependencies securely without compromising on application usability.

2.2.3 Load- and store-address dependencies

Another way for information to flow in a system is through dependencies on the memory addresses used for loads and stores. For example, consider the C code in Figure 2.1 for converting data read over the network (`InputArray`) from one format into another using a lookup table (`LookupTable`) and storing the result in another buffer (`Converted-Array`).

Each byte of input from the network is used as the offset in an address for looking up converted values in the lookup table. It is important to taint the value that gets loaded if the calculated address is tainted, because an attacker could send data over the network that would no longer be tainted after conversion but would still be data in the system that had come from the attacker. An example of how this can lead to attacks not being detected by DIFT schemes is the Code Red worm. Most existing DIFT schemes ignore load-address dependencies, so the UNICODE decoding routine that caused the buffer overflow vulnerability exploited by Code Red also removes the taint mark of the network input before overwriting the structured exception handler on the stack. Thus control flow is hijacked, but existing DIFT schemes will not detect the attack. Minos [34, 36] tracks load-address

dependencies for 8- and 16-bit loads and is able to stop the Code Red buffer overflow, but does not track 32-bit load-address dependencies. No existing DIFT scheme has successfully addressed load-address dependencies in the general case.

Store-address dependencies are also important. Without tracking these dependencies, by tainting the stored value for any store where the address is tainted, indirect attacks are possible. In addition to the possibility that an attacker could use multiple levels of pointer indirection to subvert a DIFT scheme [33, 38, 78, 89], testing has shown that even attacks not designed specifically for DIFT schemes can hijack control flow undetected due to store-address dependencies [36], such as the ASN.1 bit string heap corruption vulnerability in Microsoft Windows [106].

No existing DIFT scheme for general purpose computers can sufficiently track information flow through load- and store-address dependencies. The reason is that, even for normal systems that are not vulnerable and not under attack, information flow from the network into heap and stack pointers is common. Using heuristics to determine which address dependencies are acceptable and which constitute avenues of attack is infeasible because of the complexity of dynamic memory allocation in general purpose applications. Fortunately, our results show that, due to the relatively simple and regular memory management of sensor devices, all load- and store-address dependencies can be tracked in sensor network applications. No modifications to the application, program binary, compiler, or operating system are required. Thus, the only dependency that remains to be addressed in our scheme is control dependencies.

2.2.4 Control dependencies

Because our scheme fully tracks all four of the aforementioned dependencies, the only way for an attacker to cause information to flow from the network to control data is through control dependencies. A control dependency occurs when the value of one piece of data

affects the execution path of the program, which in-turn affects the value of another piece of data. As an example, consider the following C code:

```
if (x == 1)
     y = 1;
else
     y = 0;
```

Clearly, information flows from x to y but there is no explicit assignment and no address dependencies between these two variables. In confidentiality settings, this is referred to as an *implicit flow*. Methods exist for tracking implicit flows dynamically, including temporarily tainting the program counter on conditional control flow transfers [44, 43] and annotating the machine code with label transformations [95], but these methods are not practical without modifying the compiler or performing static analysis and binary rewriting. Furthermore, DIFT is an integrity policy that differs from confidentiality settings in that information *should* be allowed to flow through control dependencies if it does not allow the attacker to corrupt control data. Network services should read data over the network and perform different actions based on the data read, so some amount of information flow from the network to the control flow of the program is always necessary.

Where control dependencies become a security problem for DIFT is when it is possible for the attacker to *launder* the taint bit, so that a value under their control is no longer tainted. An example of this is the following:

```
UntaintedData = 0;
while ((TaintedData--) != 0)
     UntaintedData++;
```

After the above code is executed, the value of `TaintedData` will be copied into `UntaintedData` but the latter will not be tainted in the process, so that data from the

attacker is now marked as trusted. An example of a *benign* control dependency that cannot be exploited by the attacker for control data attacks is the following:

```
if (TaintedData == 0)
     HandleRequestTypeA();
else if (TaintedData == 1)
     HandleRequestTypeB();
else
     HandleRequestTypeC();
```

In this case, a conditional control flow transfer is made based on untrusted data from the network, but an attacker can only choose which path to take by sending a different kind of request over the network. Thus the attacker has only as much control over the control flow of the program as a regular client.

Another difference between confidentiality and integrity settings is that typically the former assumes that the code being executed is chosen arbitrarily by the attacker, whereas in most integrity settings like DIFT the source code of the trusted application being executed is assumed not to maliciously attempt to untaint data. Nonetheless, for general purpose systems it has been shown that regular code for operations such as `printf()`-style format string conversion and other format conversions can lead to attacker-controlled data becoming untainted and leading to attacks that evade DIFT schemes [36]. In Section 2.4 we demonstrate that no similar problems exist in the eight sensor network applications that we tested. Here we define laundering as follows.

A control dependency that causes information to flow from a tainted variable x in state s to an untainted variable y in state t is benign if and only if $I(x_s; y_t) << H(x_s)$, *i.e.*, the mutual information between the two variables after the operation is much less than the entropy of the tainted variable. Thus an attacker cannot cause her inputs to be untainted

without a consequent loss of entropy, which takes away her ability to arbitrarily corrupt data. A control dependency that is not benign is said to be laundering.

Another important distinction is that of *first-order* control dependencies *vs. higher-order* dependencies[1]. When tainted data is used in a conditional control flow transfer this causes a first-order dependency in which information can flow into untainted values. If a first-order dependency were not benign then it would be necessary to taint additional data that could lead to additional higher-order dependencies, where conditional control flow transfers that would not have depended on tainted data now do. If all first-order dependencies are benign, then there is no possibility for high-entropy, laundering second-order dependencies. Thus in Section 2.4 we need only to consider first-order dependencies, because all of these are found to be benign.

Existing DIFT schemes [34, 91, 73, 32, 80] do not consider control dependencies. Minos [34, 36] handles some laundering control dependencies by treating 8- and 16-bit immediate values differently than 32-bit immediates, assuming the former to be untrusted. This is not a principled solution, but is necessary for DIFT schemes to catch certain kinds of attacks on general purpose computers such as format string attacks [85]. In our scheme for sensor network devices, we have not added any special support for control dependencies because testing and manual analysis revealed that no control dependencies in the applications we tested are laundering, *i.e.*, they are all benign.

2.3 Implementation

In this section, we describe our emulated DIFT scheme for the MICA2 platform, and the method we used to locate first-order control dependencies.

[1]The first-order control dependency means here the first control dependency in a (possibly) nested control statement.

2.3.1 Emulated DIFT scheme

We modified ATEMU (ATmel EMUlator) [79] to initiate, propagate, and check tags. ATEMU is an instruction-level emulator that emulates the AVR processor ATmega 128L that is part of the MICA2 platform. ATEMU also implements all of the needed peripheral devices, *e.g.*, CC1000 radio chip, ADC (analog to digital converter), LEDs (light-emitting diodes), and SPI (serial peripheral interface), as plug-in libraries.

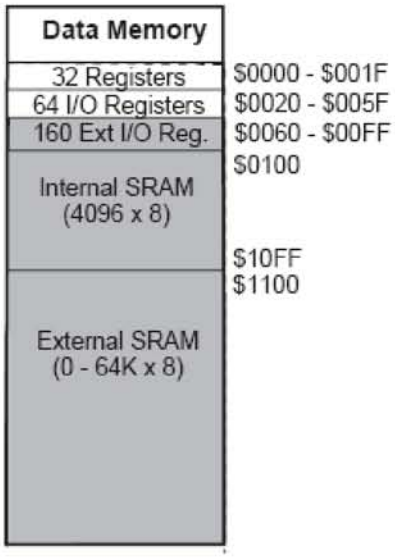


Figure 2.2: **Data memory layout, reproduced from the ATmega128 manual.**

Figure 2.2, reproduced from the ATmega128 manual [107], shows the layout of the data memory for the Atmega 128L CPU (program memory is separate). Taint marks are propagated through the registers and the SRAM. A bit-per-byte mark extension, or tag, is used for the registers and the SRAM as an indication if that location is tainted or not. All bytes in the SRAM or registers have an associated tag. Our tainting scheme can be divided into three separate functionalities: *taint source*, *taint propagation*, and *taint check*.

- *Taint source*: When the sensor device boots, all data is considered trusted. All data that is received over the network, which is the taint source in our scheme, must

Application	Main Functionality	Setup	Pass
AntiTheft	Detects and reports theft	Three nodes, a root binary and two with same binary	Yes
BaseStation	Bridges serial and radio links	Two nodes, different binaries	Yes
RadioSenseToLeds	Samples default sensor and broadcasts readings in an AM packet	Two nodes, same binary	Yes
RadioCountToLed	Maintains a 4Hz counter and broadcasts its value in an AM packet when updated	Two nodes, same binary	Yes
Oscilloscope	Samples the default sensor and broadcasts a message every 10 readings	Two nodes, same binary	Yes
BlinkToRadio	Increments a counter and sends a radio message whenever a timer fires	Two nodes, same binary	Yes
TestAM	Tests radio active messages	Two nodes, same binary	Yes
Blink	Blinks the 3 mote LEDs	One node	Yes

Table 2.1: **Application functionalities.**

be tagged as untrusted. In the MICA2 platform, the radio chip is connected to the MCU (microcontroller) through the SPI bus. The SPI connects two or more devices through a master/slave relationship. The SPI has a dedicated memory I/O register in the MCU data memory for performing data exchange. By permanently tagging this register as untrusted, all data read from the network enters the system tainted as untrusted data.

- *Taint propagation*: Our taint propagation scheme correctly handles copy, computation, load-address, and store-address dependencies. Existing DIFT schemes only consider the first two, although sometimes the latter two are handled in a limited way. For single-operand instructions such as bit rotations (*e.g.*, `ROR`) the register simply retains its tag. For instructions with multiple operands (*e.g.*, `ADD`, `SUB`, `AND`) we taint the destination register if *any* source register is tainted. This is known as a low-water-mark policy because any data that depends on tainted data is itself tainted. Register-to-register copies of data also copy the tag. `PUSH` and `POP` propa-

Application	Control dependencies	Laundering or Benign?
AntiTheft	`_vector_17()` (root)	Benign
	`runTask()` (root)	Benign
	`_vector_17()`	Benign
	`print_packet()`	Benign
	`findIdx()`	Benign
	`routingTableFind()`	Benign
	`updateNeighborEntryIdx()`	Benign
	`getOption()`	Benign
	`runTask()`	Benign
BaseStation	`_vector_17()`	Benign
	`setWakeup()`	Benign
	`putData()`	Benign
	`runTask()`	Benign
	`_vector_20()`	Benign
RadioSenseToLeds	`_vector_17()`	Benign
	`receive()`	Benign
	`runTask()`	Benign
RadioCountToLed	`_vector_17()`	Benign
	`receive()`	Benign
	`fired()`	Benign
	`runTask()`	Benign
Oscilloscope	`_vector_17()`	Benign
	`receive()`	Benign
	`runTask()`	Benign
BlinkToRadio	`_vector_17()`	Benign
	`receive()`	Benign
	`runTask()`	Benign
	`setLeds()`	Benign
TestAM	`_vector_17()`	Benign
	`runTask()`	Benign
Blink	*N/A*	*N/A*

Table 2.2: **Control dependencies in the applications tested.**

gate the tag of the register or memory location that is pushed or popped onto or off of the stack. Load (`LD`) and store (`ST`) operations apply the low-water-mark policy not only to the source data that is loaded or stored, but also to the address used for the load or store (which is always stored in the `X`, `Y`, or `Z` registers). Thus if the source of a load or store is untainted but the address used is tainted, the destination will

always be tainted so that load-address and store-address dependencies are correctly handled. Conditional control flow transfers do not taint the program counter because all sensor network applications that we tested had only benign control dependencies.

- *Taint check*: Our scheme checks the tags of target addresses (whether they are explicit operands or unstacked inputs) for all control flow transfers with target addresses that are not hardcoded as immediate values. This includes `RET` (subroutine return), `RETI` (interrupt return), `ICALL` (indirect call), `EICALL` (extended indirect call), `IJMP` (indirect jump), and `EIJMP` (extended indirect jump). If the target address is tainted, then this indicates an attempted attack and the node is reset.

2.3.2 Locating first-order control dependencies

The AVR instruction set has conditional control flow transfers that work in a way that is similar to those of the Pentium instruction set architecture. Some instructions, *e.g.*, `CMP` (compare) and arithmetic expressions, set various flags to indicate properties of their arguments such as equality, sign or less-than, overflow, *etc.* Conditional control flow transfer instructions such as `JE` (jump if equal) check these flags and either jump to a hardcoded jump target or simply move on to the next instruction depending on the value of the flag.

To locate all first-order control dependencies, we modified the emulator as follows. When an instruction sets a flag, the tag of the flag is set using a low-water-mark policy based on the operands of the instruction. When a conditional control flow occurs based on a flag that is tainted, and therefore based on untrusted data, the program counter for the location of that conditional control flow transfer is recorded. Using a variety of tools it is possible to locate the corresponding nesC code.

```
//The receiver code
10.  void overflow(uint8_t *rcm)
20.  {
30.    char buf[3];
40.    uint16_t *ptr[4];
50.    strcpy(buf, rcm);
60.    /* copy receive()'s stack pointer into caller()'s
stack pointer */
70.    *ptr[0] = *ptr[1];
80.    /* copy receive()'s ret address into caller()'s
ret address*/
90.    *ptr[2] = *ptr[3];
100. }
110. void caller(uint8_t *rcm)
120. {
130.   ...
140.   overflow(rcm);
150.   ...
160. }
170. event message_t* Receive.receive(message_t* bufptr,
void* payload, uint8_t len)
180. {
190.   uint8_t *rcm = (uint8_t*) payload;
200.   ...
210.   caller(rcm);
220.   ...
230. }
```

Figure 2.3: **Receiver code containing a vulnerability with load- and store-address dependencies.**

2.4 Results

Our experimental methodology sought to answer three key questions with respect to applying DIFT to sensor network devices and applications:

1. Is it possible to track load- and store-address dependencies, in addition to the copy and calculation dependencies tracked by conventional DIFT systems, without modifying the application, compiler, or operating system in any way? Our results demonstrate that the answer is yes.

```
//The sender code
10. payload = (char *) call Packet.getPayload(&packet, NULL);
20. strcpy(payload, "\x22\x33\44\xd9\x10\xe0\x10\xd7\x10\xde\x10");
30. call AMSend.send(AM_BROADCAST_ADDR, &packet, 11);
```

Figure 2.4: **Sender code with the attack.**

2. Is it necessary to add special taint propagation rules to handle control dependencies? Our results indicate that the answer is no, since all first-order control dependencies in the applications we analyzed are benign.
3. Does our scheme detect the attacks that it is designed to detect, including attacks with load- and store-address dependencies? Our results confirm that it does detect control data attacks.

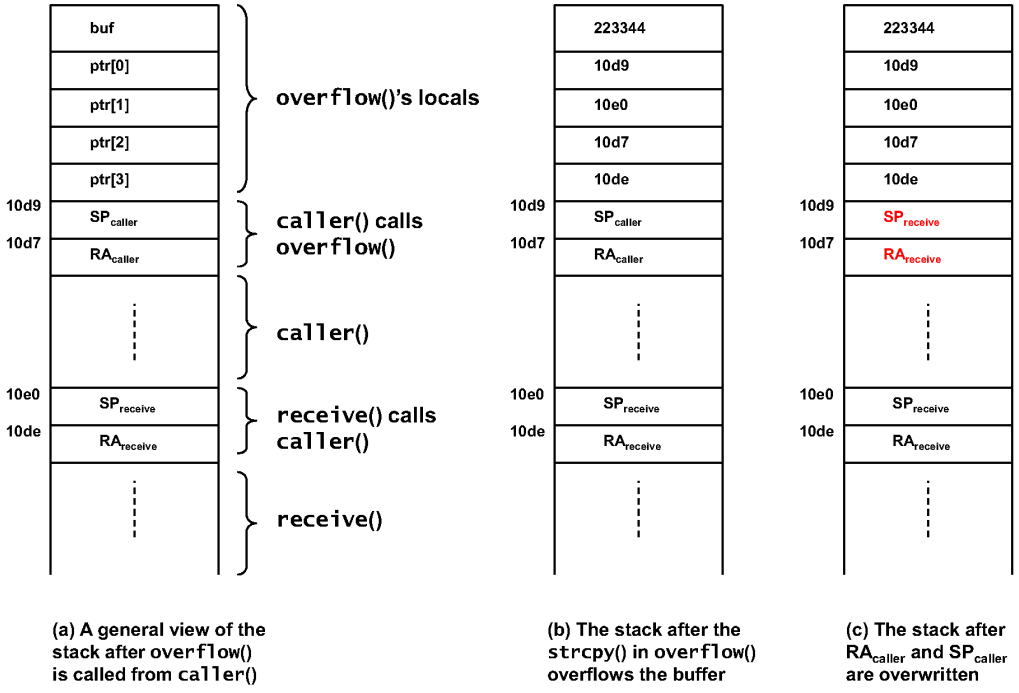


Figure 2.5: **Stack trace during the attack.**

To answer the first question we tested all applications that we were able to obtain that could be executed on the ATEMU emulator. While our hardware DIFT scheme can secure any application for any operating system on the MICA2 platform without any software

modifications, we were only able to run TinyOS [9] applications on the ATEMU emulator due to limitations of the emulator (other OSes either are not available for the MICA2 platform or do not work with the ATEMU framework). TinyOS is by far the most commonly used operating system for sensor network applications. The eight applications that we tested use a wide variety of TinyOS' functionalities, as shown in Table 2.1. Table 2.1 also shows that rigorous testing of all of the applications' functionalities yielded no false alerts. The third column describes the network setup for the application, and the fourth column indicates that all applications passed rigorous testing without producing any false alerts.

We answered the second question through a combination of testing the applications to locate the first-order control dependencies and then manual analysis of the source code for these control dependencies to determine if they are laundering or benign. The second column of Table 2.2 lists all first-order control dependencies in each application for the functionalities tested. We determined this by printing an alert with the program counter every time a conditional control flow transfer was made using a test of tainted data. As can be seen in the third column, all first-order control dependencies in all applications were determined to be benign through manual code inspection. This means that, for all applications tested, no special mechanism is needed to propagate tags for control dependencies and thus no higher-order control dependencies need be considered.

The third question is more of a claim to be confirmed than a question, since tracking all possible data dependencies will stop any attack that overwrites control data with untrusted data from the network. Whereas existing DIFT schemes measure "false positives" and "false negatives," our scheme for sensor network nodes makes satisfactory security guarantees that ensure that no remote control-flow hijacking attack based on memory corruption of control data can succeed. Nonetheless, we tested various attacks against our scheme to confirm that an alert was indeed raised in all cases. The attacks we tested include all attacks described by Francillon *et al.* [46] and Gu *et al.* [50], as well as an attack

we developed to test the tracking of load- and store-address dependencies.

The vulnerability illustrated in Figures 2.3 and 2.4 demonstrate how load- and store-address dependencies can be exploited to arbitrarily copy untainted data from one place to another. No previous DIFT schemes can stop an attack of this kind. Basically, four pointers on the stack are overwritten by the attack, then used as the source and destination addresses of two copy operations. Any DIFT scheme that does not consider address dependencies will copy data from an arbitrary source address to an arbitrary destination, both chosen by the attacker, without tainting the result. This can lead to attacks not detected by these DIFT schemes [33, 38, 78]. We tested our DIFT scheme with such an attack based on the vulnerability in Figure 2.3 and an alert was raised, demonstrating that our scheme is secure with respect to address dependencies.

Figure 2.5 (a) shows the layout of the stack on the receiver side after receiving the message coming from the sender and then executing the following sequence of function calls: `receive()` calls `caller()`, and then `caller()` calls `overflow()`. The stack shows how the local variables and activation records (including return addresses and stored stack pointers) are laid out on the stack. Figure 2.5 (b) shows the exact values that will occupy the local variables of `overflow()` right after line 50 is executed. Note that the `strcpy()` in line 50 has overflowed the buffer `buf` with more data than it has room to store. The extra data overflows the four pointers right after the buffer `buf`. Figure 2.5 (c) shows the stack contents after the execution of lines 70 and 90. The stack pointer of `caller()` has been changed to that of `receive()`, and the return address of `caller()` has been changed to that of `receive()`, Note that the return address has not been overwritten by `strcpy()`, but changed using a tainted pointer. Now when `overflow()` returns it returns to `receive()` directly, bypassing the `caller()` function, and will continue functioning as normal. Existing DIFT schemes will not detect this attack because they do not track load- and store-address dependencies. We tested this attack on our scheme and it was indeed detected. While the vulnerability in Figure 2.3 is

a hypothetical example, vulnerabilities with load- and store-address dependencies are not uncommon in real application code, as discussed in Section 2.2.

2.5 Discussion and Future Work

In this section, we discuss four aspects of our DIFT scheme for sensor network nodes that could be explored further in future work.

2.5.1 Scaling up to other embedded devices

While the focus of this chapter has been on sensor network nodes, an interesting question is what other systems can DIFT schemes that check load- and store-address dependencies scale up to. Researchers have explored address dependencies on general purpose systems [89], finding that the complex memory management of these systems makes tracking these dependencies infeasible. It is an open question whether a DIFT scheme that tracks address dependencies would be practical on other embedded devices with more complex memory management than sensor network nodes. Many false alerts could be ameliorated by untainting tainted data under certain conditions. We leave this as the subject of future work.

2.5.2 Non-control data attacks

It is well known that attacks can hijack control flow or otherwise take control of a machine without overwriting control data [27]. After control data and low-level control flow is secured then this can serve as a foundation for securing other types of data. Furthermore, it would be possible to extend our DIFT scheme to enforce various type systems by increasing the number of bits and allowing for more complex taint propagation rules. The

Elbrus line of machines as described by Babayan [16] implemented this type of security for C programs, where any memory corruption attack that is a type violation (virtually all are) is prevented, but general purpose applications must be rewritten for reasons related to memory management. Whether sensor networks can be secured against non-control data attacks in this fashion and the impact this would have on power and performance are questions left for future work.

2.5.3 Control dependencies and other types of information flow

Because all first-order control dependencies in the applications that we tested were benign, we did not implement any special taint propagation rules for control dependencies. For applications that do have laundering control dependencies it would be possible to give developers an automated way of finding these and a secure way to taint them, perhaps by annotating their source code.

Because we are enforcing an integrity policy, rather than confidentiality, we assume that it is safe to ignore covert timing channels [59, 63, 100].

2.5.4 Hardware modifications and power and performance overheads

DIFT can be implemented simply by adding an extra bit to every byte in the registers, pipeline, and memory, and some simple logic gates where operations occur [91, 34]. The performance and power overheads are negligible. While any pipeline modification requires adoption of a DIFT scheme by chip manufacturers, several reasons make it more likely that sensor network applications will benefit from DIFT than general purpose computers. These include the following:

- One reason that hardware DIFT support has not been adopted by chip manufacturers

for general purpose computers is that for these systems DIFT provides no real security guarantees. In contrast, we demonstrate that satisfactory guarantees of security against a major class of attacks can be achieved for sensor network applications.

- The economics of embedded devices presents less barriers to hardware support for security, testing, reliability, *etc.*, as evidenced by successful extensions such as JTAG [53].
- The MCU and memory of sensor network nodes are often packaged as one device so only one chip needs to be changed, whereas proper DIFT support for general purpose computers would require new DRAM chips, a motherboard with extra bits for sockets and buses, and operating system support for virtual memory swapping with tag bits.

2.6 Related Work

We divide related work into two parts: worm attacks and defenses in sensor networks, and security against remote memory corruption.

2.6.1 Worm attacks and defenses in sensor networks

Gu *et al.* [50] showed that remote control data attacks can be exploited on sensor network nodes, and Francillon *et al.* [46] demonstrated that after exploiting a buffer overflow an attacker can even reprogram instruction memory with the SPM instruction. Our DIFT scheme for sensor network applications stops any attack that corrupts control data.

Ferguson *et al.* [45] propose a defense against control data attacks on sensor nodes that is similar to Control Flow Integrity [10, 11]. This incurs significant power and performance overhead and requires modifications to the program binary. Furthermore, attacks are not detected after control flow is hijacked until the attacker executes some code that

transitions to another function. This is because the control flow sequence check is performed at the end of each function. Furthermore, the function marks are stored on the stack for function calls and are therefore also subject to corruption, and because they are 8 bits in length they can also be guessed with a brute force attack. Lastly, interrupt routines are not considered in the control flow sequence.

Kumar *et al.* [58] use software-based fault isolation to provide a coarse-grained form of memory protection for sensor applications and the operating system. Yang *et al.* [101] present a defense that is based on software diversity, where multiple versions of an application are created. Regehr *et al.* [82] implement efficient type and memory safety on a small, 8-bit embedded device. This requires modifications to the entire compilation chain, and even after optimization some applications can have more than a 35% increase in code size and a 6% performance overhead. Our proposed DIFT scheme would require hardware modifications, but would work with existing compilers and binaries with no need for binary rewriting, and have negligible power and performance overhead.

2.6.2 Security against remote memory corruption

The two research areas most relevant to our own regarding security against remote memory corruption are DIFT schemes for general purpose machines, and control flow integrity.

DIFT was introduced by Suh *et al.* [91]. Crandall and Chong [34] and Crandall *et al.* [36] explored various policy tradeoffs for DIFT schemes and higher-level systems issues such as virtual memory swapping, and Vigilante [32] employed DIFT for automated worm defense. Further research focused on making DIFT more flexible either in software [73] or through more flexible hardware extensions [39]. Argos [80] is a widely-deployed honeypot technology based on DIFT. Researchers have also analyzed the security and applicability of DIFT [33, 38, 78, 89].

Our scheme is the first DIFT scheme for sensor network applications. As discussed

earlier in this chapter, no existing DIFT schemes for general purpose computers address load-address, store-address, or control dependencies in a way that supports satisfactory security guarantees against control data attacks. Our scheme tracks all load- and store-address dependencies, and through testing and manual analysis we determined that all sensor network applications that we tested had only benign control dependencies. While flexible DIFT schemes [39, 73] could allow any policy to be enforced, nobody has actually specified a DIFT policy that secures general purpose systems. Research suggests that this is impractical for general purpose systems due to address dependencies [89], and even existing attacks not intended to evade DIFT have been shown to not be detected by existing DIFT policies [36].

Control Flow Integrity (CFI) [10, 11] uses binary rewriting and checks to ensure the validity of all control flow transfers. CFI requires no hardware modifications, and gives strong security guarantees even in an attack model where the attacker can read or write any location in memory. However, CFI requires significant changes to the compiler and library linking infrastructure. Furthermore, an implementation of something similar to CFI for sensor network nodes showed significant performance and power overheads, as well as a large increase in code size [45]. Without further research, it is not clear if these overheads were due to the particular implementation or are inherent to CFI.

DIFT for other things besides control data attacks...

2.7 Conclusion

We have presented a DIFT scheme for sensor network applications that is secure against remote control data attacks because all data dependencies are either tracked or shown to be benign. Whereas existing DIFT schemes for general purpose computers consider only two of the five data dependency types (copy, calculation, load-address, store-address, and

control), our scheme tracks the first four and testing and manual analysis of applications demonstrates that the fifth is always benign (for the applications we tested, other sensor network applications could easily be tested in a similar manner). Testing of eight applications also showed that no modifications to the compiler, operating system, application source code, or program binaries is necessary. We expect that, whereas DIFT has not yet led to secure general purpose systems, DIFT will lead to secure sensor network devices with security built-in "from the ground up."

Chapter 3

Secure and Usable DIFT System for Intrusion Detection

Current intrusion detection systems (IDSes) fall into two very limiting categories: appearance-based or behavior-based. IDS operators specify attacks that should be detected either by some type of signature or statistical invariant, or by specifying a sequence of operations (*e.g.*, system calls) that is considered malicious. What if we could specify intrusions as flows of information? This would enable the IDS systems of the future to very explicitly define attacks in a general way that precluded entire categories of vulnerabilities and exploits.

The main reason why IDS systems do not currently have this ability is that dynamic information flow tracking (DIFT) technology is extremely limited in its current form. DIFT works by tagging (or tainting) data and tracking it to measure the information flow throughout the system. Existing DIFT systems have limited support for address and control dependencies, and therefore cannot track information flow within a full system without addressing these very challenging dependencies in an ad-hoc fashion.

In this chapter, we present a prototype DIFT system that supports address and control

dependencies in a general way. As a motivating example to demonstrate this system, we define an attack by the amount of control that external network entities have over what a networked system is doing. We measure the amount of information flow between tainted sources and the control path of the CPU for a variety of scenarios and show that our prototype system gives intutive, meaningful results.

3.1 Introduction

Information flow is a fundamental concept in computer and network security. Dynamic information flow tracking (DIFT) systems could enable a wide variety of applications, but their applicability is currently very limited because important information flow dependencies are not tracked for stability reasons. We define *stability* of a DIFT system to mean that the amount of taintedness in the system should not increase unless the amount of information in the system from a tainted source has increased. Without this property the entire system quickly becomes tainted and nothing can be learned about the actual information flow. We define *accuracy* to mean that the measured information flow should be close to the real information flow in the system.

In aircraft design, a technique called "relaxed static stability" allows for the design of aircraft with advanced maneuverability and stealth capabilities by relaxing the requirement of designing the aircraft to have inherent positive stability during flight. Modern fighter jets and stealth aircraft are designed with inherently negative stability, then advanced digital "fly-by-wire" systems are incorporated into the design to create a stable system that can actually fly. This has opened up possibilities for aircraft design that were thought to not be possible before. In this chapter we apply this same general principle to the design of DIFT systems, and demonstrate that this makes it possible to track address and control dependencies in a stable DIFT system, something that existing DIFT techniques have not been able to achieve with a general method. While our current DIFT system is based on an

open-loop controller, *i.e.*, there is no feedback from the system output back into the control system, approaching DIFT as a control problem and decoupling the competing design requirements of stability and accuracy enables stable, full-system dynamic information flow tracking.

Current DIFT systems that are stable achieve this only by ignoring large classes of information flow or treating them in an application-specific manner. Existing DIFT systems [91, 34, 73, 32, 80] are based on the general notion of tracking "taint" marks. A taint mark, which can be a single bit, is associated with every byte or word in the memory. Data from some particular source is "tainted" and these taint marks are propagated and used to approximate information flow. For example, in the application of DIFT to detecting control flow hijacking attacks based on memory corruption, all bytes that come from the network are tainted. These taint marks are propagated so that as tainted data moves around in the registers and memory of the system and new data is calculated based on tainted inputs, any data based on the untrusted source (typically the network) is tainted. Then memory corruption attacks that hijack control flow can be detected by checking all jumps, calls, and returns to ensure that their destination addresses are not based on tainted data [91, 34, 73, 32, 80].

Systems that measure information flow and do track address and/or control dependencies are either unstable [43, 44, 95, 93] or have limited, application-specific support for these dependencies. Panorama [103], for example, demonstrates the power of full-system information flow measurement, but handles address and control dependencies in an application-specific manner (see Section 3.6 for details).

Suh *et al.* [91] divided information flow into five types of dependencies that DIFT systems must track to be complete: copy dependencies, computation dependencies, load-address dependencies, store-address dependencies, and control dependencies. Copy dependencies are movements of the data from register to register, register to memory, or memory to register. The obvious way to track these dependencies is to also copy the taint

mark from source to destination. Computation dependencies are operations on the data such as, *e.g.*, when two registers are added and the result is stored in a destination register. Most DIFT systems are conservative and taint the result if either of the sources were tainted.

All existing practical DIFT systems, including those intended for applications other than detecting memory corruption attacks, either ignore or provide very limited support for the last three types of dependencies. Address dependencies during loads or stores would require that the destination of the load or store be tainted if the address used is tainted. For example, consider the following C code for converting an array of tainted input from one format to another using a lookup table:

```
for (Loop = 0; Loop < Size; Loop++)
    Converted[Loop] = LookupTable[Input[Loop]];
```

In terms of real information flow, the value stored in `Converted[Loop]` should be tainted if the value loaded from `Input[Loop]` is tainted. This will only be true for a given DIFT system if the DIFT system checks the taintedness of the address used for the load with `LookupTable` as its base and propagates this taint. DIFT systems do not do this in the general case because it causes instability in the DIFT system where everything in the system quickly becomes tainted. Store address dependencies are a related problem for stores instead of loads. One of the contributions of this work is a DIFT system that does track address dependencies in a general way.

The DIFT system we present in this work is also able to track control dependencies, whereas previous DIFT systems have not tracked control dependencies, except in limited, application-specific ways. Control dependencies are related to the classic problem of implicit information flows, and arise from information flows such as the following from `x` to `y`:

```
if (x == 1)
     y = 1;
else
     y = 0;
```

Address and control dependencies are practical concerns for all DIFT systems [91, 89] and have prevented the application of DIFT outside of very restricted domains. Many remote attacks, such as script code injection or Trojans embedded in PDF documents, require both of these dependencies to be tracked in some way in order to be detected. Thus DIFT-based honeypots have only been deployed for detecting remote control flow hijacking attacks based on overwriting control data [34, 35, 80]. Also, consider the application of a practical data provenance system that keeps track of fine-grained information flow within a system, where the threat model is not an attacker who can write arbitrary code to leak information, but rather the accidental leak of confidential information, *e.g.*, when a user cuts and pastes from a word processor file into a presentation and then saves the file to external storage. If the external storage is lost, the organization would like to have a detailed, fine-grained record of what confidential information was on it. Because of format conversions (*e.g.*, ASCII to UNICODE or object-oriented clipboard format conversions) and other practical considerations there will be many address and control dependencies that must be tracked for the DIFT system to work as intended.

As mentioned previously, current DIFT systems cannot track these dependencies in a general manner because tainting address and control dependencies causes instability in the DIFT system. If you simply propagate taint marks for either of these dependencies conservatively, every object in the system quickly becomes tainted, which tells us nothing about the information flow in the system. Outputs of interest will be tainted whether or not there was real information flow to them. This has been identified as a major limitation for DIFT [91, 89], and is not solved by information flow tracking systems designed for confidentiality, such as Fenton's Data Mark Machine [43, 44], RIFLE [95], or GLIFT [93],

because they also "overtaint" without static analysis, which is impractical for most DIFT applications.

In this chapter, we address this problem by building a DIFT system that is not inherently stable because it *does* track address and control dependencies, and then add an open-loop control system that throttles how sensitive the DIFT tracking is to these dependencies. In our prototype relaxed static stability DIFT system, taint values are fixed point numbers rather than single bits so that we can make approximations about the amount of information flow (*i.e.*, a byte can be tainted with 8.0 bit of taintedness, or 0.125 bits of taintedness, *etc.*). The tags represent the mutual information between the data object the tag is associated with and the taint source. **Relaxed static stability will enable new applications that are impossible with traditional information flow controls or current DIFT systems.**

Other than the following two assumptions, there are no limitations to the possible applications of relaxed static stability DIFT:

1. The attacker either cannot execute arbitrary code or does not know about the DIFT tracking. We do not claim that our DIFT system can track information flow if arbitrary code is written with the express purpose of dropping the taint marks. This is a very difficult problem, and many applications do not need such a strong threat model.

2. Approximations of the true information flow are good enough so long as they capture the most important aspects of the information flow. In other words, as long as the output tells the user of the DIFT system something meaningful about the true information flow in the system, perfect accuracy from an information-theoretic perspective, which is impossible to achieve in practice, is not a requirement.

In this chapter, we present a relaxed static stability system that is based on an open loop control system and use it to measure information flow from various taint sources

into the control path of the CPU. The control path of the CPU includes what instructions are executed, their operands, the data that conditional control flow decisions are based on, and the addresses instructions are fetched from—these basically encompass what the CPU is doing at any given time. While such a system could be used for detecting attacks, *e.g.*, in a similar spirit to Newsome *et al.*'s concept of undue influence [72], we make no claims in this work in regards to supporting or refuting a claim that measuring information flow into the control path of the CPU dynamically is a viable detection technique for remote network attacks. Newsome *et al.* demonstrate that a static approach based on the general idea of measuring this flow of information can distinguish attacks from false alerts. Our results show that relaxed static stability can take meaningful measurements of full-system information flow into the control path of the CPU dynamically, whether or not this dynamic approach is useful specifically for detection in terms of false positive and false negative rates is beyond the scope of this work. We chose this application only because it is challenging and the flows of information are intuitive so that we can compare measurement results to our intuitions.

Our claims are the following:

- **Address and control dependencies are both critically important to track in practical applications of DIFT.** This has been pointed out in previous work [91, 89] but always within the context of detecting control data attacks. Our results demonstrate that, especially when DIFT is applied to more general applications and real systems, address and control dependencies are the rule of whole-system information flow, not an exception.

- **A relaxed static stability approach to DIFT enables information flow measurements that are sensitive to address and control dependencies while still remaining stable.** This is achieved because the competing design requirements of stability and accuracy are decoupled, so that accurate information flow measurement in

future work will only be limited by the research community's imagination in designing control systems. Our results demonstrate that, even with a relatively simple open loop control system, accurately measuring full-system information flows (that existing DIFT systems are not capable of measuring due to address and control dependencies) *is* possible.

3.2 Implementation

The first step in building our prototype was to generalize the notion of a tag from a single taint bit to a fixed point number representing how tainted a data object is. In our system, every byte of the registers and memory has a 16-bit fixed point number associated with it, which represents how much tainted data is in that byte (measured in bits of entropy in an information theoretic sense from 0.0 to 32.0). We chose this range because sometimes in our DIFT system the taintedness of a 16-bit or 32-bit word is stored in its least significant byte. These tags constitute a 200% storage overhead, which is also an issue for performance, but we discuss how this can be ameliorated in future work in Section 3.5.

When every byte in the system is augmented with a tag in this way, we can define taint propagation rules and a source and sink of information flow and measure the amount of information flow over time between the source and sink. Note that all of the tag propagation occurs at the architectural level in a virtual machine, based on raw bytes and machine instructions. This means that full-system information flow measurement is possible and no modification to the operating system or software being measured is necessary.

For our preliminary study, we define the source of tainted data to be the network (*i.e.*, data from the attacker), and the sink to be the control path of the CPU (which instructions are fetched and what data is used for control flow decisions). By measuring the information flow over time between this source and sink we can approximate how much

control the attacker has over what the CPU is doing, even if their inputs are high-level shell commands or script code. This allows for a very general definition of attack: the more information flow from the network into the CPU control path, the more likely it is that the system has been attacked. This definition of attack, unlike existing DIFT-based honeypot technologies [34, 80], makes no assumptions about how the attacker gained control of the system, *i.e.*, it does not matter if the exploit is a buffer overflow, script injection, a guessed password, or any other type of remote intrusion.

To implement the source and sink, we taint all data that comes from the network card of the virtual machine (with 8.0 bits of taint for a byte, 16.0 bits for a word, and 32.0 bits for a double word). Using a moving average filter, we plot the amount of tainted data that is used in the control path of the CPU (fetched instructions, data used for conditional control flow decisions, and control data).

Our prototype is built on top of Argos [80], which is itself built on top of the QEMU emulator for the x86 architecture, and handles the five dependencies defined by Suh *et al.* [91] as follows:

- **Copy dependencies:** When data is copied from register to register, memory to register, or register to memory its taint tag is also copied from source to destination. For load instructions, a throttle constant of 0.99 is multiplied by the loaded taint value and then stored in *both the destination and source*. Note that address and control dependencies can also affect copy operations.
- **Computation dependencies:** When an operation is performed on one or more source operands and stored in a destination, the maximum taint of the source operands is stored in the destination taint tag. Note that address and control dependencies can also affect computation operations.
- **Load and store address dependencies:** If the address of a load or store operation, when multiplied by a constant address dependency throttling factor of 1.0 (*i.e.*, we are not currently throttling address dependencies), has a taint value that is greater

than the source (or maximum of the sources), then the taint value of the address, instead of the taint value of the data, is copied to the destination's taint tag.

- **Control dependencies:** When tainted data is used for conditional control flow decisions (*e.g.*, `x` or `y` in the C code "`if (x == y)`" that will be compiled into a compare instruction followed a conditional jump in assembly), the program counter is tainted with the maximum taint of the values compared. This is implemented by tainting condition flags for compare instructions, and checking the taint of flags used in conditional control flow transfers. If the program counter is currently tainted and its taint, when multiplied by another throttle factor, is greater than that of the taint value of the source of a copy or computation operation, the destination is tagged with the higher taint value instead. This makes tracking of control dependencies possible. The throttle factor for the program counter is currently set to the constant 0.5, but could be varied dynamically in future work with a closed loop controller. If a tainted flag is used and its taint value is copied to the program counter, the taint value of the flag is multiplied by the constant throttle factor of 0.99 for reasons explained below. For every instruction where the control flow is not based on tainted data the program counter's taint level is multiplied by 0.99. The amount of taintedness in the CPU control path is calculated for each instruction as a moving average y of the program counter's taintedness e, using the equation $y' = cy + (1 - c)e$ with $c = 0.999511719 = \frac{2047}{2048}$. (This is the y-axis for all figures in Section 3.4.)

Also, address dependencies are only enabled when the CPU is not in supervisory mode, *i.e.*, these dependencies are only applied in user space. This is satisfactory for most applications, but with more advanced control systems tracking these dependencies in the kernel space will also be possible if necessary. Control, copy, and computation dependencies are tracked throughout the entire system, including the kernel.

3.2.1 Design decisions

The above design decisions are the result of extensive testing to develop a controller that was both stable and sensitive to the information flow being measured. The load throttle constant of 0.99, which is applied to the taint of loaded values in memory where they are stored, was necessary because in our experiments many values are stored to memory once and then read many times, becoming endless sources of taintedness. We found that forcing the taintedness of values that sit in memory to decay over time is essential for system stability. By having a high value for this constant, it only affects these problematic values in a significant way and normal data that has a shorter lifetime is not significantly affected. The other constants (0.5 for the program counter throttle, 0.99 for conditional control flow loop throttling, and 0.99 for the program counter decay) were also the result of extensive testing to make the system as sensitive as possible without becoming unstable. The value of $c = 0.999511719$ was chosen as the highest value for c, *i.e.*, the one that keeps the most history in each iteration, that could be represented in our fixed point format which uses 5 bits for the whole part and 11 bits for the fractional part.

Our current prototype uses an open loop control system, based on a moving average filter and multiplying by constant throttle factors. Also, it uses the maximum of multiple taint values, instead of addition. From an information theoretic point of view, the maximum operation assumes the maximum joint entropy between tainted variables while addition assumes zero joint entropy. The actual amount of information flow will typically be somewhere in between these two extremes. Still, because of the power of the relaxed static stability approach, our prototype produces very good results when measuring the information flow between the network and the CPU control path.

In building our prototype, we discovered the importance of loop structures in building relaxed static stability DIFT systems. Consider the example of a control dependency from Section 3.1:

```
if (x == 1)
        y = 1;
else
        y = 0;
```

In our DIFT scheme, the taint value of x will be copied to the program counter when the `if` condition is checked, and then transferred from the program counter to y (and multiplied by the program counter throttle of 0.5) when the value of y is set. This has the desired effect from an information flow perspective, and is similar in concept to Fenton's Data Mark Machine [43, 44]. Now consider the following loop where Y is an array.

```
while (x > 0)
{
        x = x - 1;
        Y[x] = 1;
}
```

Ignoring the address dependencies, we can see why the condition check taint values need to be throttled. If $x = 1000$ and the tag of x is tainted with 3.5 bits of information, then without throttling each element of the array Y will be tainted with 3.5 bits. This means that from 3.5 bits of tainted information we will have generated 3500 bits of tainted information. This obviously violates the law of the conservation of information in a problematic, unbounded way. By throttling the taint value of the condition flags for x each time they are checked by multiplying by 0.99, we can ensure that, in this example, at most $\frac{1}{1-0.99} \times H(x) = 100 \times H(x)$ (in this case, 350) bits of taint is generated in Y by the conditions on x. This is still more entropy than existed before the loop but is bounded by a geometric series. Because of the many subtleties of information theory, any practical DIFT system is an approximation, including ours. For future work in accurate DIFT systems based on relaxed static stability we anticipate that the information theory of looping constructs [65] will be of critical importance.

3.3 Experimental methodology

We tested our system with a variety of attacks and in other scenarios involving high-level languages, and also using controlled rates. Our experimental methodology was designed to answer the following questions:

1. **Are the measurement results of our DIFT system repeatable?** Because of CPU scheduling, time dilation in the virtual machine, and other factors measurements are not deterministic. They should be repeatable in the sense that any measurement result is representative of a ground truth, however. We repeated a particular attack (the Code Red worm) ten times to ensure this.
2. **Are address and control dependencies critical to measuring information flow?** We disabled address and control dependency tracking in our DIFT system, both together and individually, for all attacks to assess this for our motivating application.
3. **Do the information flow measurements of our DIFT system match intuitions about what the actual amount of information flow is?** While defining a ground truth for the actual amount of information flow based on information theory to compare our measurements to is infeasible due to the various subtleties of information flow, if the measurements show what is going on in the system in a meaningful way then the measurement results are accurate enough to be valuable in many applications. We tested attacks for this purpose, and we also tested information flow in controlled situations.
4. **Does tracking address and control dependencies allow us to measure information flow into the CPU's control path even when the tainted code is in a high-level language?** To answer this question, we tainted Dhrystone binary or source code in four formats: compiled binary code (from C), Java byte code, Perl code, and Python code.

In Section 3.4, our results answer all of these questions in the affirmative. We also took the following steps to ensure that the interpretation of our results was clear:

- We compare the information flow from source to sink (*i.e.*, network to CPU control path) during attacks to the information flow for the same exact network request against a non-vulnerable version of the system. In other words, we repeat the attack against a patched version of the system. This ensures that the differences between an attack and its baseline are due to the attacker taking control of the system, and not due to differences of protocol usage and other system factors.
- For repeatability reasons, we disabled automatic updates when necessary, and performed all measurements against an idle system.

Table 3.1 shows the eleven attacks that we tested our DIFT system with. Though several of them were exploited by high-profile worms (*e.g.*, 8205 for Blaster, 5310 for Slammer, and 10108 for Sasser) we tested all but one of them using publicly available exploits that simply spawn a remote shell that the attacker can subsequently connect to. The exception is the Code Red worm, based on the IIS Server ISAPI buffer overflow (2880), which we tested with the Code Red worm itself.

Note that only a few of the attacks we tested are simple buffer overflows where information flows directly from the network to a piece of control data, which is the type of attack that traditional DIFT systems have been able to catch [91, 34, 73, 32, 80]. The ASN.1 heap corruption vulnerability and the IIS Server ISAPI buffer overflow both have address dependencies because of format conversions. The format string vulnerability in Nullsoft SHOUTcast has both address and control dependencies, as do all format string vulnerabilities. The two web browser vulnerabilities are both attacks that are initiated when the victim visits a web page controlled by the attacker, and the first phase of the attack is JavaScript code to spray the heap [98]. We purposely chose a diverse variety of attacks for testing to demonstrate the power of relaxed static stability DIFT to measure

Vulnerability	bugtraq ID [106]
ASN.1 Integer Overflow	9633
ASN.1 Heap Corruption	13300
DCOM RPC Buffer Overflow	8205
Nullsoft SHOUTcast Format String	12096
MS-SQL Resolution Service Heap Overflow	5310
MS-SQL Authentication Buffer Overflow	5411
LSASS Buffer Overflow	10108
NetDDE Buffer Overflow	11372
Internet Explorer MPEG2TuneRequest	35558
Firefox 3.5 TraceMonkey	35660
IIS Server ISAPI Buffer Overflow	2880

Table 3.1: **Attacks we tested our DIFT system with.**

information flow in general. Recall that we make no claims about the application of these results to detection with good false positive and false negative rates. Our aim is to show the potential of information flow to revolutionize the way we think about IDS and shed light on research directions that can make this happen.

3.4 Results

In this section we present results to support the two main claims in regards to our DIFT prototype, that tracking address and control dependencies is both: (1) necessary for meaningful measurements of the information flow in a real system, and (2) made possible by a relaxed static stability approach.

3.4.1 Graph legend and axes

All graphs in this section, unless otherwise noted, have the following axes and legend. The x-axis is time in increments of 50 milliseconds, *i.e.*, 100 on the x-axis is equal to 5

seconds. Note that time can be dilated by the performance overhead of the virtual machine. The y-axis is equivalent to the rate of information flow from source to sink (in something proportional to bits per second), so can be interpreted as the amount of tainted information from the network that is flowing into the control path of the CPU at that time.

Time dilation is a factor since our moving average is applied on a per-instruction basis while the value itself is sampled (for the graphs) based on time. Another caveat is that we have applied an additional moving average filter of $c = 0.99$ to all of the graphs in this section to make them more readable. Where more than one line appears in a graph, the lines are independent tests plotted approximately on the same axis for comparison. Where one or more lines are omitted from a graph, it is because they were effectively zero and indistinguishable from the x-axis.

Results of tests against vulnerable versions of operating systems and services follow the legend shown in Figure 3.1. Solid black lines are tests where we utilize the full power of relaxed static stability DIFT, and track all dependencies (including address and control dependencies) as described in Section 3.2. We also sought to assess the importance of address and control dependencies individually. Solid grey lines are tests where control, copy, and computation dependencies are tracked but address dependencies are disabled by ignoring the taint of addresses for loads and stores (all other taints propagate in the same way). Black dotted lines are tests where address, copy, and computation dependencies are tracked but control dependencies are disabled by setting the control dependency throttle to 0.0 instead of 0.5.

Grey dots are tests where only copy and computation dependencies are tracked, and both address and control dependencies are disabled. Modulo two caveats, this can be viewed as how a typical standard DIFT system [91, 34, 73, 32, 80] would perform. The two caveats are that we still apply a load throttle of 0.99 to loaded values in this case, and the moving average filter has the effect of smoothing out small bursts of measured information flow.

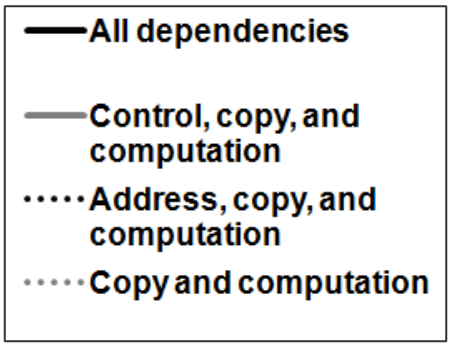


Figure 3.1: **Legend for vulnerability results.**

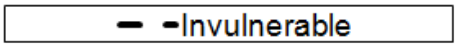


Figure 3.2: **Legend for invulnerable baseline results.**

Figure 3.2 shows the legend for results of tests against invulnerable (*i.e.*, patched) versions of operating systems, while tracking all dependencies including address and control. This means that the test uses the same protocols as the tests against vulnerable versions, but the exploit where control flow is taken over by the attacker does not succeed. This serves as a baseline to compare an attack to that ensures that the differences are only related to the success of the attack and not the protocols involved. These tests are represented with a dashed black line.

3.4.2 Basic results

Figure 3.3 shows the result of the Code Red worm attacking our DIFT system running a vulnerable version of IIS web server and Windows 2000. Recall that for Code Red, unlike the other tests in this section, the exploit we used is the actual worm itself rather than an exploit that binds a shell to a port and goes to sleep. It is clear that our DIFT system is able to measure the rapid increase in information flow from the network to the

control path of the CPU when the worm takes control of the machine. The falloff at the end occurs when all 100 threads that Code Red spawns complete their initial TCP/IP requests and go into a sleep state waiting for remote victims to respond with SYN/ACK packets. From this figure it is also clear that both address and control dependencies, and in particular their interactions together, are a critical part of this. All three of the other tests, where address or control dependencies or both are disabled, are indistinguishable from the baseline invulnerable case shown in Figure 3.4.

Figure 3.4 shows the same network traffic of a Code Red infection, but directed at an invulnerable web server (patched by installing service pack 4 for Windows 2000). The same initial bump, from about $x = 50$ to $x = 150$, can be seen in both figures. This is the initial HTTP protocol processing where the web server is processing a request that contains malformed UNICODE encodings. The difference between the two graphs is that in the invulnerable version control flow is never hijacked by overwriting a structured exception handler on the stack, meaning the web server returns a `404 page not found` and closes the connection and the worm never remotely gains control of the machine.

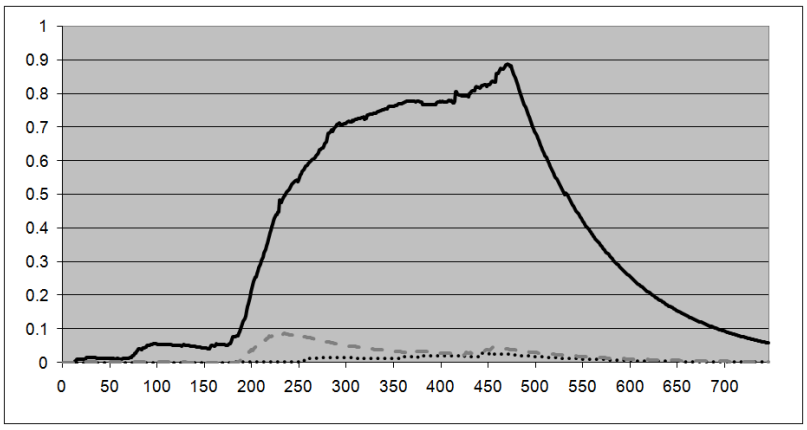


Figure 3.3: **Code Red attacking a vulnerable version of Windows 2000 and IIS web server.**

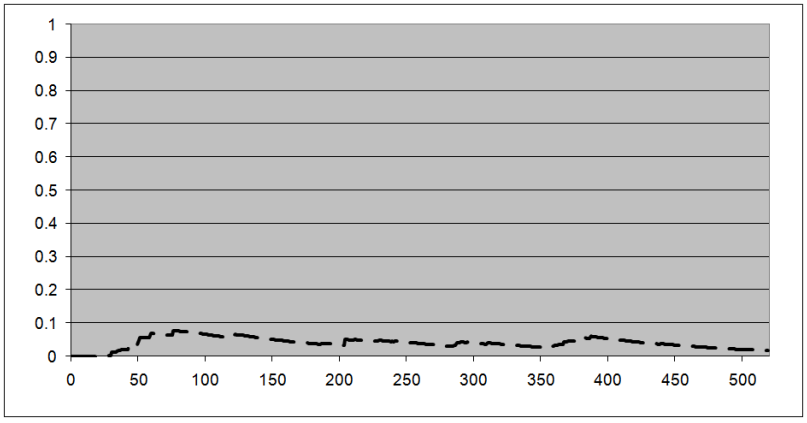


Figure 3.4: **Code Red attacking an invulnerable version of Windows 2000 and IIS web server.**

3.4.3 Repeatability

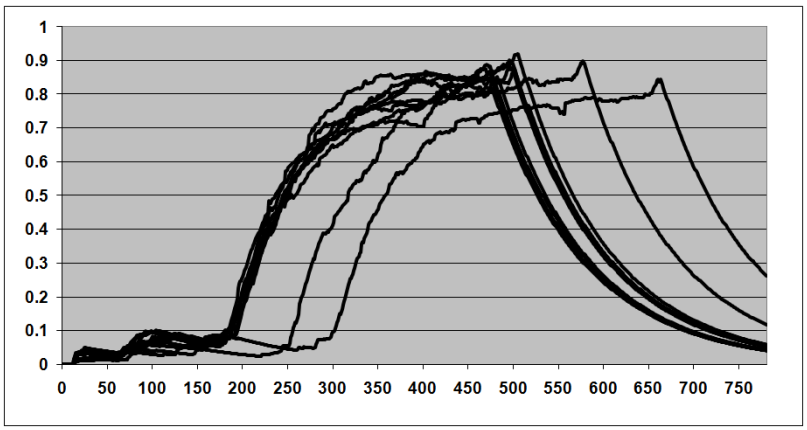


Figure 3.5: **Code Red attacking a vulnerable VM, repeated.**

Figures 3.5 and 3.6 show the same tests as Figures 3.3 and 3.4, respectively, repeated ten times in each case. Due to time dilation of the virtual machine and system nondeterminism due to process/thread scheduling, we repeated these tests to demonstrate that the graphs produced by our DIFT system are repeatable and representative of the same ground truth for a given test. All of the results in this section that we repeated had this property.

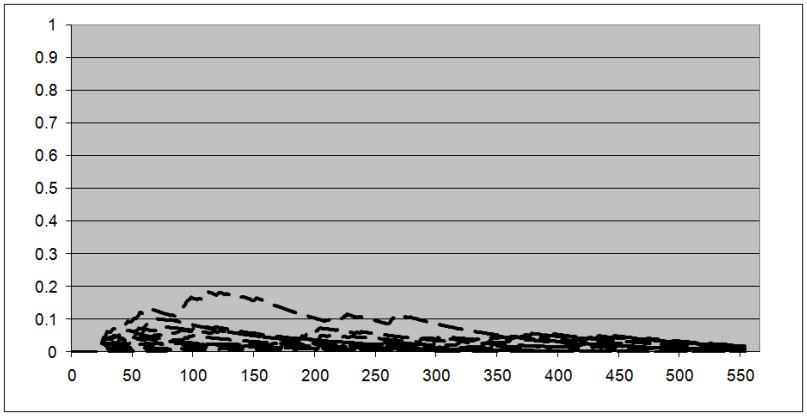


Figure 3.6: **Code Red attacking an invulnerable VM, repeated.**

3.4.4 Importance of address and control dependencies

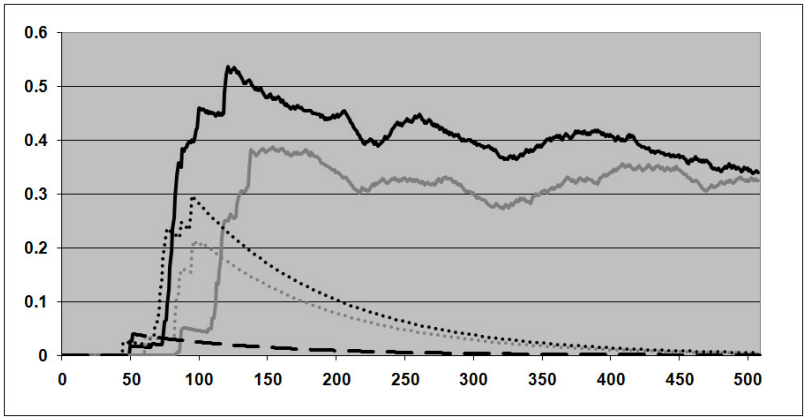


Figure 3.7: **9633 (ASN.1 Integer Overflow).**

All of the tests for other exploits that we tested our DIFT system with support our conclusions that address and control dependencies are important and that relaxed static stability DIFT can track these dependencies accurately. Each exploit proved to have unique features as well.

Note that the graphs for the various exploits do not have the same scale on the x- and y-axes. Our prototype DIFT system was built to demonstrate the concept of relaxed static

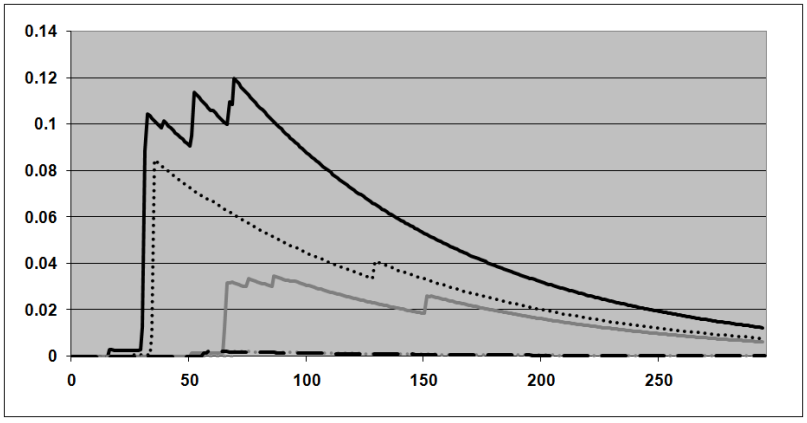


Figure 3.8: **13300 (ASN.1 Heap Corruption).**

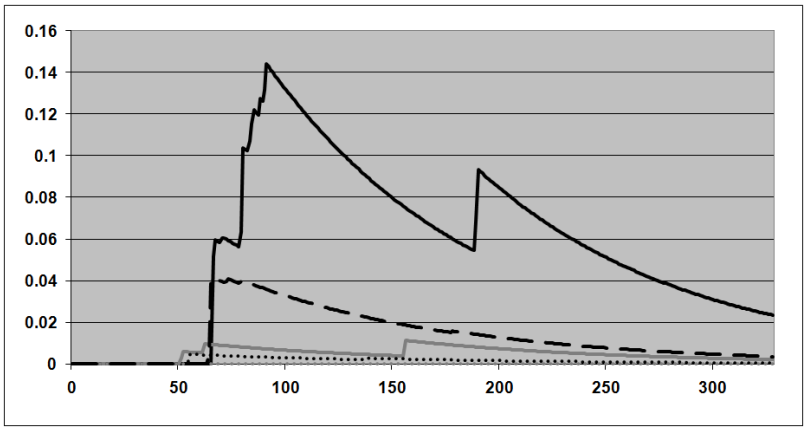


Figure 3.9: **8205 (DCOM RPC Buffer Overflow).**

stability DIFT and the importance of tracking address and control dependencies. To be used for something like detection in a honeypot scenario, other application-specific concerns would need to be accounted for, in particular the difference of scale between various exploits. In this work we use the honeypot application as a vehicle for demonstrating relaxed static stability DIFT in an intuitive way, we make no claims about being able to apply our current DIFT system as a honeypot in a real environment with good false positive and false negative rates.

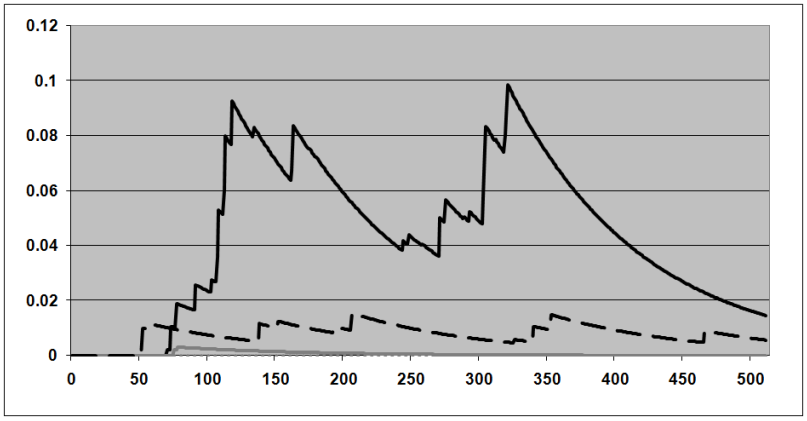


Figure 3.10: **12096 (Nullsoft SHOUTcast Format String).**

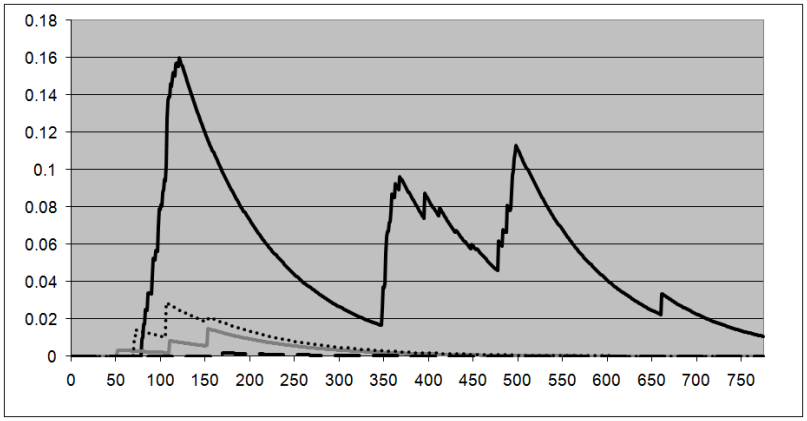


Figure 3.11: **5310 (MS-SQL Resolution Service Heap Overflow).**

Figures 3.7 and 3.8 show the results for exploits for two different vulnerabilities in the ASN.1 parser in Windows. It is interesting that Figure 3.7 shows control dependencies as a major driver of the information flow while Figure 3.8 shows the same for address dependencies. ASN.1 is an XML-like format for binary data, so various encodings and lookup tables are involved when decoding the raw bytes that come from the network. Another interesting note is that the exploit for 13300 is one that conventional DIFT systems aimed at detecting control data attacks [91, 34, 73, 32, 80] will not detect when control flow is hijacked because the exploit overwrites a pointer to a function pointer, not the

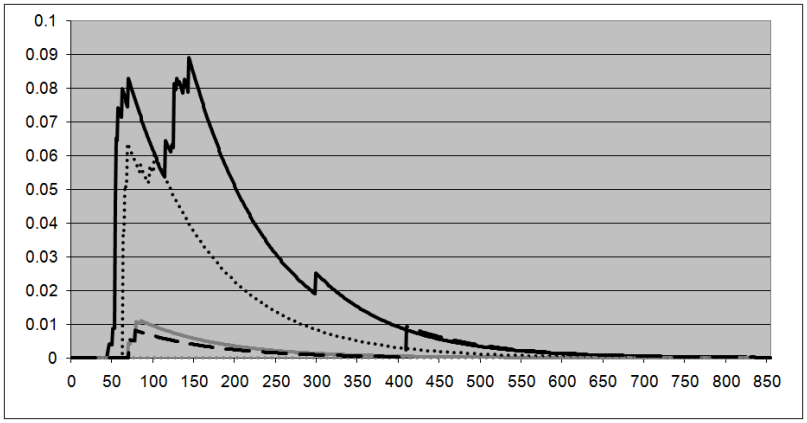


Figure 3.12: **5411 (MS-SQL Authentication Buffer Overflow).**

function pointer itself, meaning that there is a 32-bit address dependency in the corruption of the control data.

Figure 3.9 shows test results using the exploit that was used by the Blaster worm. Without both address and control dependencies, the measured information flow is actually less than that of the invulnerable test, meaning that it might not be possible to detect this attack without tracking *both* address and control dependencies, and certainly would not be possible without tracking either. This also shows that it is not just a matter of tracking both address and control dependencies, but the interdependencies between address and control dependencies are important as well. Figure 3.10 shows the same, and Figures 3.11 and 3.14 also show results where the vulnerable and invulnerable tests are indistinguishable unless both address and control dependencies are tracked. Figure 3.12 is interesting in that the information flow is driven almost entirely by address dependencies.

There are two interesting aspects to Figure 3.13. This exploit is the same exploit that was used by the Sasser worm. To exploit the buffer overflow in Windows' LSASS service, the attacker opens a TCP/IP connection directly to the Windows kernel and sends various commands to open a named pipe, write some data into the named pipe, *etc.* On an unpatched Windows XP machine with no service packs, this is not in itself an attack but is

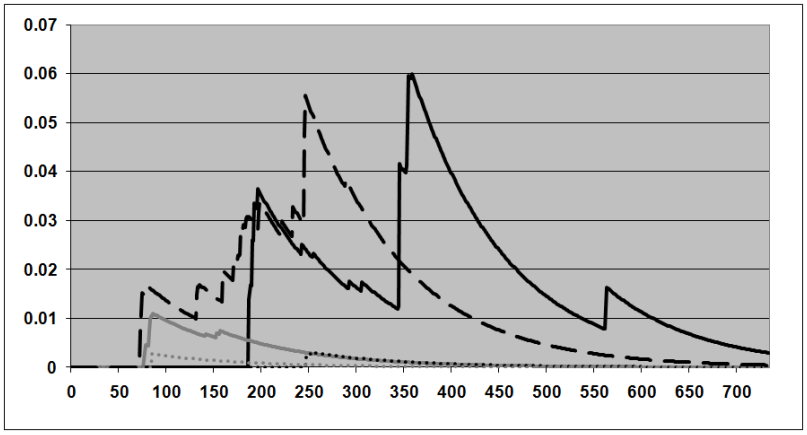


Figure 3.13: **10108 (LSASS Buffer Overflow).**

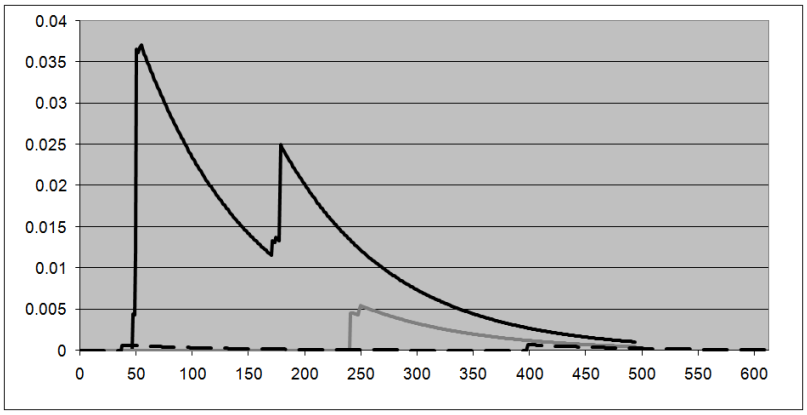
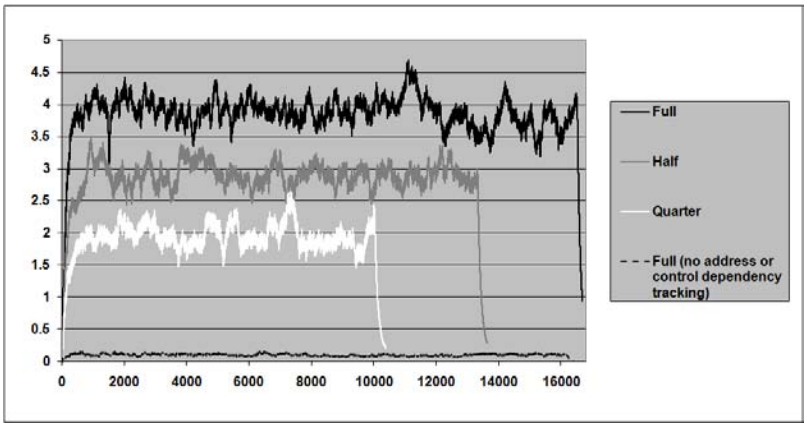
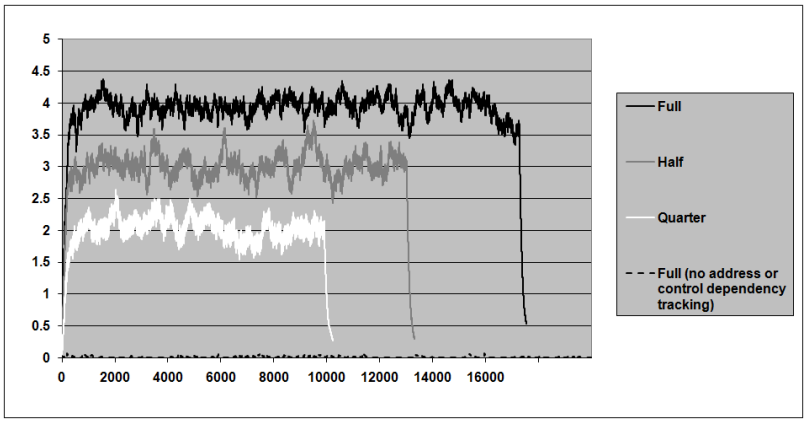


Figure 3.14: **11372 (NetDDE Buffer Overflow).**

actually a feature that Windows allows. Thus, the attacker does have a large amount of control over the control path of the CPU even before overflowing the buffer, so it is not surprising that the vulnerable and invulnerable tests are indistinguishable in terms of magnitude. One distinguishing factor is that the vulnerable case shows two phases (instructing the kernel to do various IPC operations, and then executing arbitrary code) whereas the invulnerable version has only one phase (instructing the kernel to do various IPC operations). Another distinguishing feature is the extra increase in information flow (*i.e.*, the "bump" in the vulnerable test at about $x = 550$). Through repeated tests we confirmed



(a) Normal



(b) Compressed

Figure 3.15: **Measured information flow for controlled rates.**

that this bump is when the attacker connects to the shell that has been bound to a TCP/IP port, and represents an increase in control because the attacker has initiated a command shell. If the attacker types any commands into this shell, there is a significant increase in the measured information flow to the control path of the CPU.

3.4.5 Controlled rates

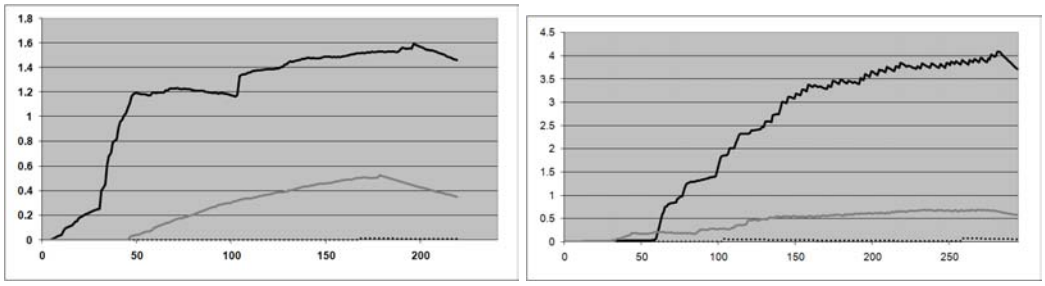
We also tested our DIFT system's measurements with controlled rates to determine if the measurement outputs matched the rates that we set. For these experiments we used the CD-ROM as the taint source and repeatedly copied different compiled dhrystone binaries (to avoid the effects of disk caching) from the CD-ROM to a RAM disk and executed them. For the normal uncompressed case, we fixed the rate at which we did this at 0.8 repititions of this per second, 0.4 repetitions per second, and 0.2 repetitions per second, corresponding to the "Full," "Half," and "Quarter" experiments in Figure 3.15(a). Figure 3.15(b) is the compressed experiments that were set at $\frac{2}{3}$ per second for "Full." The full rates were different because the unzipping of the Dhrystone binary takes some extra time. We modified the Dhrystone binary to perform fewer loops per execution so that each execution takes between 1 and 2 seconds.

For the first set of experiments, shown in Figure 3.15(a), we simply copied each raw binary and executed it. For the second set of experiments, shown in Figure 3.15(b), the binaries had been compressed using gzip on the CD-ROM so they were copied, uncompressed to introduce additional address and control dependencies, and then executed. For both sets of experiments, the rates measured generally corresponded to the controlled rate in terms of relative magnitudes. There was a small effect of time dilation that made the measured values not exactly proportional. Table 3.15(a) shows the integral rates over time, normalized to the full rate, which is closer to proportional. These results show that, although the measured values of our DIFT system have no meaningful units without calibration, the measurements can be calibrated to the actual amount of information flow in bits per second. Without tracking address and control dependencies, Figures 3.15(a) and 3.15(b) show some information flow is measured in the simple case but the information flow is barely perceptible without tracking the address and control dependencies in gzip.

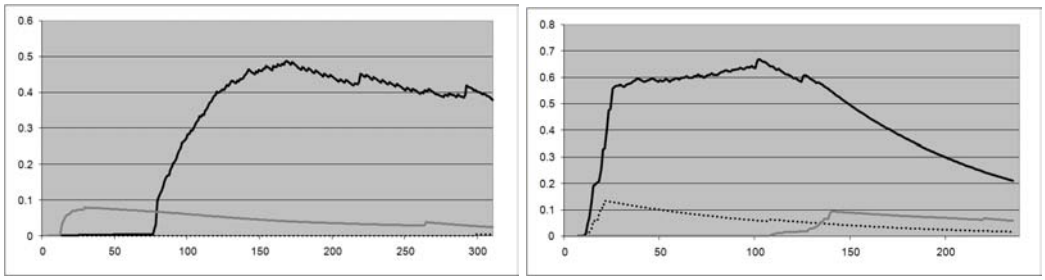
Actual Rate (Controlled)	Uncompressed Measurement (Figure 3.15(a))	Compressed Measurement (Figure 3.15(b))
Full (1.0)	1.0	1.0
Half (0.50)	0.60	0.58
Quarter (0.25)	0.31	0.30

Table 3.2: **Rate measurement integrals over time, normalized to full rate.**

3.4.6 High-level languages



(a) C on the left, Java on the right.



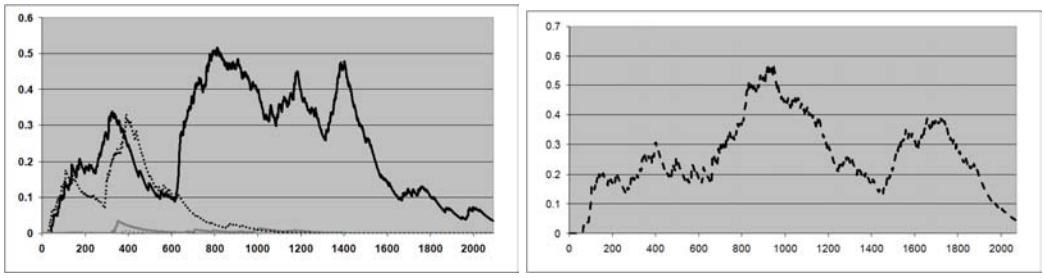
(b) Perl on the left, Python on the right.

Figure 3.16: **High-level languages.**

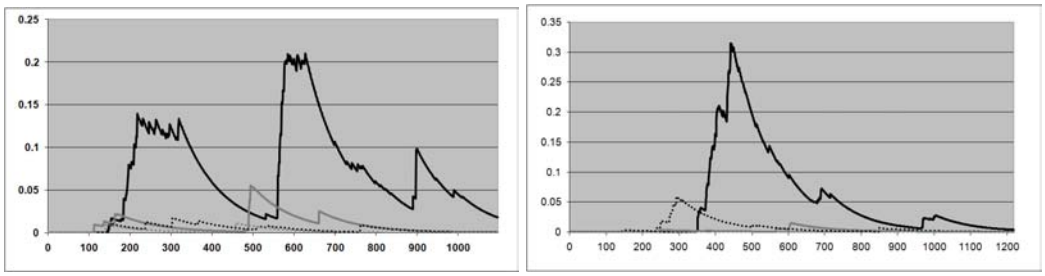
Another important question to answer is if relaxed static stability DIFT can measure information flow from a tainted source into the control path of the CPU for arbitrary code, even when that code is in a high-level interpreted language. Any DIFT system that does not track address and control dependencies will not detect this information flow because

there are many table lookups and conditional control flow transfers involved when any interpreter or just-in-time compiler is parsing and executing a high level language. Figures 3.16(a) and 3.16(b) show the results of tracking tainted Dhrystone code in compiled binary format (from C), Java byte code, Perl, and Python, respectively. Clearly, address and control dependencies are important for high-level language execution, and relaxed static stability DIFT performs well in all four cases.

Browser exploits



(a) Firefox 3.5 on the left, Firefox 3.5.1 on the right.



(b) Internet Explorer 6 on the left, Internet Explorer 7 on the right.

Figure 3.17: **Heap-spraying browser exploits.**

In July 2009, two browser vulnerabilities were announced: one for Internet Explorer and one for Firefox. The Firefox vulnerability was fixed in Firefox version 3.5.1, while the Internet Explorer vulnerability existed in both Internet Explorer 6 and Internet Explorer 7. The results of testing these four browser versions against their respective exploits, with Firefox 3.5.1 being invulnerable, are shown in Figures 3.17(a) and 3.17(b). The interesting

thing about these graphs is that the information flow from the network into the control path of the CPU after the browser visits a web page with the exploit is largely due to heap spraying [98] and not the exploitation of the vulnerability itself. Thus it represents a high system load due to Javascript that has been crafted by the attacker.

3.5 Discussion and future work

We anticipate that relaxed static stability DIFT will open up new research possibilities for security applications and benefit from future advances in accuracy and performance due to the application of more advanced control theory. The key to future work on relaxed static stability DIFT will be the accuracy *vs.* performance tradeoff. More accurate information flow measurements for a variety of applications will be possible with dynamic feedback control systems [51], but storing a fixed point number for every byte in the system will be prohibitive for the performance of some applications.

Hardware support is one way to increase performance, which can be integrated into the processor core [91, 34], and designed for flexible policies [39, 81] or decoupled on a separate coprocessor [55]. Many of these systems perform some form of compression to reduce storage overhead and memory bandwidth usage. On-demand emulation [52] has been shown to yield near-native performance for some workloads, and performance improvements for DIFT on commodity hardware have been demonstrated by parallelizing the security checks on multicore systems [74]. Many of these performance enhancements exploit the fact that large portions of memory are untainted. It is not clear if this applies to relaxed static stability DIFT systems, since the tag represents a degree of taintedness and many objects in the system will be tainted, at least slightly, by address and control dependencies. One possible approach to achieving a form of compression for relaxed static stability DIFT would be to perform a form of interpolation similar to that performed by graphics processors, *e.g.*, taint tags could be calculated based on density fields rather

than stored in and loaded from a large tag memory. Another possibility is probabilistic tagging, where a single tag bit or small number of tag bits is associated with each byte, and probabilistic tag propagation is used to approximate the information flow.

While we do not claim to have built a DIFT system that, in its current form, detects a wide variety of attacks in a general way with low false positives and false negatives, our results show that this approach can be very promising **if DIFT technology addresses a few specific challenges in future research.** The fact that we were able to measure information flow in a manner that is both stable and matches our intuitions about the attacks tested means that intrusion detection via information flow measurements is possible. If the research community builds a new foundation for information flow research that is focused on integrity, availability, and intrusion detection then we can succeed where confidentiality systems of the past failed. This is because the precision that confidentiality applications require is not necessarily required in other settings. As Salvador Dali said, "Have no fear of perfection, you'll never reach it."

3.6 Related Work

DIFT in the context of detecting attacks was introduced by Suh *et al.* [91]. TaintBochs [28] was another early application of tainting that was applied to analyzing data lifetime in a full system. The Minos project [34, 36] explored various policy tradeoffs for DIFT schemes and higher-level systems issues such as virtual memory swapping, TaintCheck [73] explored some of the issues of using DIFT for end-to-end detection of exploits for vulnerabilities in commodity software, and Vigilante [32] employed DIFT for automated worm defense. Argos [80] is a widely-deployed honeypot technology based on DIFT. DIFT has been applied to problems outside the domain of detecting control data attacks, such as malware analysis [103, 102, 18], network protocol analysis [62, 99], and dataflow tomography [69]. This shows that DIFT is a general technique with a broad range of applications,

but a relaxed static stability approach to DIFT will greatly multiply the possible applications, since address and control dependencies are so prevalent in so many applications.

The Panorama project [103] demonstrated the power of full-system dynamic information flow tracking. However, the way that address and control dependencies are handled in Panorama is application specific. A control dependency in the UNICODE conversion of keystrokes was handled via manual labeling. Also, all load address dependencies are propagated in Panorama so that ASCII to UNICODE conversions and other table lookups are handled appropriately. This propagation rule was stable for the applications Panorama was applied to, but is not stable in general [36, 89], especially when combined with store address dependencies.

The earliest DIFT papers [91, 34] identified the problems with address and control dependencies. More details and analysis of these issues followed [33, 26, 38, 78], including a recent quantitative analysis of full-system pointer tainting [89]. Flexible tainting schemes that allow taint policies to be specified [91, 73, 97, 81, 39, 31] often allow for address and control dependencies to be incorporated into the policy, but do not fundamentally address the stability issue. Policies written for these systems that incorporate address dependencies usually tradeoff accuracy for stability by ignoring other dependencies such as computation dependencies. Although it is possible to *specify* policies that track all dependencies in these schemes, the overtainting problem will lead to a system where everything is tainted. **Based on the empirical evidence of five years of DIFT research, we believe that no stable, conventional DIFT system can accurately track both address and control dependencies in a general way. This is our motivation for proposing a relaxed static stability approach.**

Dynamic information flow systems that have confidentiality as their primary goal [43, 44, 95, 93] have the same overtainting problem. This is because they are based on a conservative notion of confidentiality as a noninterference property. This is necessary for, *e.g.*, multi-level secure systems, but for many practical confidentiality concerns such

as data provenance and accidental leaks of information noninterference is not necessary. Overtainting in confidentiality systems is due to the fact that, within the context of non-interference, information *does* flow throughout the system in a pervasive manner. This is compounded by the fact that, without some form of static analysis, implicit flows must be handled conservatively by tainting every object in the entire system.

Fenton's Data Mark Machine [43, 44] requires a special instruction set to define when the program counter can be untainted, and also requires that all registers in the system be statically assigned as tainted or untainted. Thus, a practical compiler for Fenton's system would need to know the taint value of every piece of data at every possible program point, which implies that the compiler has already identified all control dependencies through static analysis. RIFLE [95] was applied to the Itanium instruction set, but in the absence of assertions about possible implicit flows (provided by static analysis) RIFLE will basically taint every data object in the system on a conditional control transfer. GLIFT [93] makes strong guarantees of noninterference at the bit level, but translating this into a practical information flow tracking system will not be possible without static analysis. For all of these systems, in the absence of static analysis to make assertions about code that does not execute, every conditional control flow transfer must taint every piece of data in the system.

Decentralized information flow control (DIFC) [68] is a concept based on information flow that relaxes the requirement of noninterference while still making confidentiality and integrity guarantees. Several systems [40, 105, 57] have been based on DIFC or similar ideas. These full-system information flow tracking systems are similar in spirit to DIFT, but the applications and implementations of DIFC are fundamentally different from DIFT and are largely based on manual declassification. DIFC systems work at a higher level of abstraction than DIFT systems. While DIFC systems can track information flow for a full system, they do not address the problem that DIFT systems have of dealing with address and control dependencies.

Newsome *et al.* [72] present a method for detecting attacks that is based on measuring information flow into the control path of the CPU. Their approach is a purely static approach, for static approaches methods for dealing with address and control dependencies are known. The only connection Newsome *et al.* has to DIFT is that they use DIFT traces as a loop-free program for input to their static analysis.

McCamant and Ernst [66] propose a technique for quantifying information flow that is not based on tainting. It combines static and dynamic analysis to observe one or more program executions and calculate a network flow capacity. TightLip [104] is a related approach, but is purely dynamic and only observes two program executions to check for non-interference.

To summarize, all existing DIFT schemes can be categorized into two extremes: either they are stable at the cost of missing important information flows, or they are unstable to the point of being entirely impractical without static analysis[1]. Relaxed static stability DIFT moves the possible design constraints beyond this dilemma and will enable new DIFT applications, in much the same way as the analogous approach in aircraft design has led to the design of aircraft with advanced stealth and maneuverability capabilities.

3.7 Conclusion

We showed that information flow has the potential to revolutionize intrusion detection systems and identified specific research challenges for dynamic information flow tracking will help us to achieve this. Our relaxed static stability DIFT system prototype gave intuitive results at measuring the information flow between the network and the control path of the CPU. This is something that could be used in a general definition of intrusion that escapes

[1]Note that "static" in the sense of static analysis means something different than in the context of stability. In the former case static means that the code is not executed dynamically for analysis, in the latter it means the system as designed, without its integrated dynamic control system.

the limitations of current appearance- and behavior-based definitions. Whereas signatures and sequences of system calls define known bad behaviors, information flow can be used to define what it means for an attacker to attack a system at a very fundamental level. We would be honored to attend NSPW and foster a discussion centered around the following thought experiment: **How would we design the IDS systems of the future if we could assume that all of our wishes for dynamic information flow tracking had come true?**

Chapter 4

Antivirus Security: Timing Channel Attacks Against Antivirus Software

Remote attackers use network reconnaissance techniques, such as port scanning, to gain information about a victim machine and then use this information to launch an attack. Current network reconnaissance techniques, that are typically below the application layer, are limited in the sense that they can only give basic information, such as what services a victim is running. Furthermore, modern remote exploits typically come from a server and attack a client that has connected to it, rather than the attacker connecting directly to the victim. In this chapter, we raise this question and answer it: Can the attacker go beyond the traditional techniques of network reconnaissance and gain high-level, detailed information?

We investigate remote timing channel attacks against ClamAV antivirus and show that it is possible, with high accuracy, for the remote attacker to check how up-to-date the victim's antivirus signature database is. Because the strings the attacker uses to do this are benign (*i.e.*, they do not trigger the antivirus) and the attack can be accomplished through many different APIs, the attacker has a large amount of flexibility in hiding the attack.

4.1 Introduction

Network reconnaissance is a vital step for the remote attacker before launching an attack. Attacking every reachable host is not desirable to the attacker because only vulnerable hosts can be successfully penetrated. Port scanning is a well-known technique that provides the attacker with very useful information about possible victims. The attacker wants to know if the victim is running certain services, and determines this by sending packets to certain ports the victim might be listening on. The communication between the attacker and victims could reveal the victims' specific services and operating system, and even version information. Port scanning, while helpful to the attacker, is limited by the kind of information it can attain and by the ability to reach victims that have contacted the attacker from behind Network Address Translation (NAT). Stateful firewalls and intrusion detection systems, if well deployed and configured, are considered strong defense lines against port scanning. Also, maintaining stealth is important. In this chapter, we explore techniques to remotely gain detailed high-level information about a victim host that has connected to the attacker's web server. Our goal is to better understand how the future network reconnaissance techniques that network defenders must anticipate will work.

4.1.1 Application-level reconnaissance

We are particularly interested in the ability of the remote attacker to learn about victims beyond traditional techniques. The highest level of information the attacker can obtain is information related to the victims' running applications, particularly security-related applications. Vulnerable applications, firewalls, and antivirus software are of interest to the attacker. We take the antivirus application in our study as a running example and show how the attacker can stealthily gain information about the victims' antivirus. In this example, the attacker creates a timing channel to infer how up-to-date the antivirus is. This could be useful in that the attacker could decide to send older versions of malicious code

and thereby limit the exposure of their newest version by only using it when necessary. In traditional network scanning, the remote attacker scans victims by the means of being a client who tries to connect to servers and discover vulnerabilities. In our model, the attacker is modeled as a server who waits for clients' connections and then scans them at the application layer. This model is particularly suited to drive-by-downloads.

4.1.2 Threat model

While modern antivirus software employs advanced techniques such as filtering, algorithmic scanning, and emulation, at its heart antivirus is still based on pattern matching. Note that the filters that trigger algorithmic scanning and emulation are still pattern-based, and advanced techniques also amortize their performance and therefore offer opportunities for timing channels.

An antivirus scanning engine scans data against its virus signatures. The antivirus does not compare data to every single signature sequentially, but rather it stores the signatures in data structures that allow for fast scanning that is optimized for the common case (typical strings of bytes) and amortizes the performance overhead by having slow code paths that are only taken when a byte string is close to a signature in the database in some way. Depending on how the antivirus stores the signatures in the data structures, scanning one piece of data can take a longer time than another based on the scanning path the antivirus takes to determine if the data is malicious or benign. Suppose that the attacker knows how the antivirus scanning works, then they can create special crafted data that makes the antivirus take a longer time if a certain signature is in the database, but less time otherwise. In our threat model, the attacker wants to know if a client's antivirus database is updated with a certain signature or not. Although the attacker is modeled as a server in our threat model, she can be modeled as a malicious insider as well.

The fundamental principle the antivirus software utilizes is making the common case

fast. However, this introduces the possibility of timing channel attacks.

4.1.3 Why antivirus?

Having antivirus software is considered essential for typical computers today. According to a study [4], 81 percent of users use antivirus on their computers. Antivirus signature databases vary widely in terms of how up-to-date they are, due to both users who have not updated recently and scaled releases of updates. An attacker need not use a more recent malicious code, and thereby increase the exposure of the more recent code, if a user's antivirus signatures are not up-to-date and an older malicious code will suffice.

We chose ClamAV antivirus in our study because it is an open source antivirus.

4.1.4 Why timing channels?

The benefit of a timing channel attack is flexibility and stealth. Even though the attacker might be able to directly check how up-to-date the antivirus' database files are through ActiveX controls or other APIs that allow direct checking of directories, files, and processes, this suspicious behavior will have a distinct behavioral signature that is difficult for the attacker to obfuscate (*e.g.*, opening the antivirus' signature database). Also, the database files could be hidden or not allowed to be reached by the attacker in the first place. So, indirectly inferring how up to date the database is is preferable from the attacker's point of view.

4.2 Background

4.2.1 On-access vs. on-demand scanning

On-access scanning is triggered upon file system operations, such as ***open***, ***create***, or ***close*** system calls. To be scanned with the on-access scanner, a virus should be read from or written to the disk. On-access scanners run as daemons and hook into the file system APIs or are implemented as device drivers that are attached to the file system [92]. On-demand scanning starts only if the user asks the scanner to scan some files.

4.2.2 ClamAV antivirus

ClamAV [2] is a well-known, open source antivirus program. ClamAV consists of a main library and a set of command line programs that make use of the APIs provided by the library. On-access scanning in Windows is possible via Clam Sentinel (see below).

File type filtering

The ClamAV scanning engine has 10 different roots which correspond to 10 different file types. These are GENERIC, PE, OLE2, HTML, MAIL, GRAPHICS, ELF, ASCII, NOT USED, and MACH-O. ClamAV signatures are loaded into the data structures of those roots depending on what kind of files a virus can infect. For example, if virus **X** infects PE files and a signature **X'** is generated for **X**, then **X'** will be loaded into the data structures of the PE root. When ClamAV scans a file, it checks the file type first to determine which root's signatures will be used to scan that file against. File type filtering speeds up the scanning process by optimizing scans for file types where the entire file need not be scanned.

Filtering step

To make scanning even faster, ClamAV implements an additional filtering step prior to scanning. Every type root has its own filter. The filter can determine if a file is benign before scanning it. The most important feature of these filters is that they do not have false negatives but do have false positive. In other words, the filter will not let a file containing a virus pass without being scanned, but if it can prove that the file is benign then no further scanning is needed for that file. However, some other benign files might cause a hit in the filter and thus will need to be scanned further. ClamAV implements a bit-level state-machine to match characters in the filter. The state machine has 8 states where each state is represented by 1 bit. The state machine might have multiple active states at the same time and thus multiple transitions might be taking place in parallel. Because ClamAV checks input against the filter, any character can be good to start checking from. Thus, ClamAV activates state 1 at each transition. An active state 1 is represented by 11111110. The filter is an array, called B, of length 65536, where each element is 8 bits long. Figure 4.1 illustrates this. ClamAV chooses 8 characters carefully from each signature to add it to the filter. Then, it iterates through the 8 characters and reads q-grams of length 2 at each position. For example, if the 8-byte string of a signature is 0x001122334455667788, then it changes the filter as in the following steps[1]:

1. For position 1 of the string, execute B[0x0011] = B[0x0011] & 11111110. This says that 0x0011 is satisfactory to start with.

2. For position 2: B[0x1122] = B[0x1122] & 11111101.

3. It continues in the same way for the following positions until the 7th position.

4. To mark the end of a string ClamAV has another array called End. For the previous

[1] $<<$, $|$, and & represent shift left, or, and and bitwise operations, respectively

example, when position 7 is reached, the End array is changed to be End[0x7788] = End[0x7788] & 10111111.

After ClamAV determines which type root an input belongs to, it checks the root's filter against the input. Searching the filter starts by setting the bit-level state machine to 1111111 (no active states). Then, ClamAV iterates through all input characters until it finds a match or reports a negative result. At each character position, ClamAV reads 16 bits as q0 (q-gram equals 2) and performs this statement:
state = (state << 1) | B[q0], where state << 1 activates state 1 each time, and then it checks if it finds a match by this statement:
match_state_end = state | End[q0], and reports a match if match_state_end != 0xff. In other words, it reports a match when reaching a state at which the input can end at while being in an active state. Because this filter is created by applying this procedure to all of the signatures, a string will not pass the filtering step unless it cannot possibly match any signature.

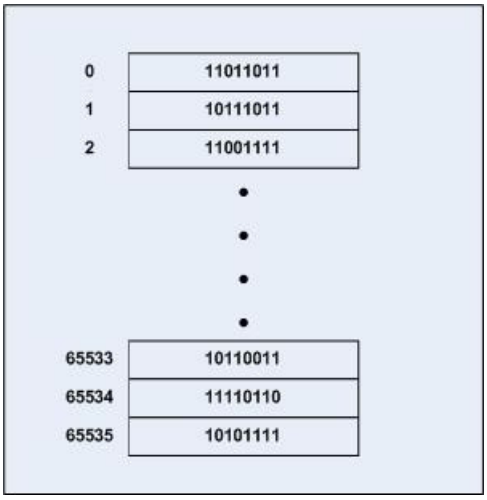


Figure 4.1: **ClamAV filter. The filter content is based on the signatures. An active position is represented by 0. The right most bit is the first position.**

Aho-Corasick algorithm

ClamAV uses an extended version of Aho-Corasick algorithm [12]. ClamAV usually uses this algorithm for signatures that have wildcards. In this case, the signature is divided into patterns that need to be matched in order to report a match. This algorithm is used to match an input against many patterns at the same time. ClamAV uses a tree-like data structure, called a trie, to store patterns. Each node in this data structure has 256 transition pointers. Each transition represents an ASCII character. What distinguishes this data structure is that each node has the same prefix as its predecessor nodes up to the root node, so that all patterns with the same prefix will take the same path. Because each node takes a considerable amount of memory and also because the trie could be very large if not restricted, ClamAV has put a maximum limit for the depth to be 3. After reaching the maximum depth, all patterns will be added to a linked list attached to leaf nodes. Suppose that there is a signature that starts with the sequence 0x010002. Figure 4.2 shows the transitions in the ClamAV trie structure that this sequence would take. Once the leaf node is reached, the pattern will be added to the linked list of patterns which have the same prefix. Also, fail transitions are established, so that instead of returning back to the root at every mismatch, a fail transition is made to the proper node. For example, if the next input at node 2 is 0x01, then the fail transition is set to go back to node 2 instead of re-matching starting again from node 1.

Boyer-Moore Algorithm

An extended version of Boyer-Moore algorithm [21] is used in ClamAV. This algorithm is usually used for signatures which do not have wildcards. The original Boyer-Moore algorithm scans patterns against input from right to left. Two tables are used to determine how many characters the pattern needs to be shifted by. The two tables are built based on two roles, the bad character shift role and the good suffix shift role. For this algorithm,

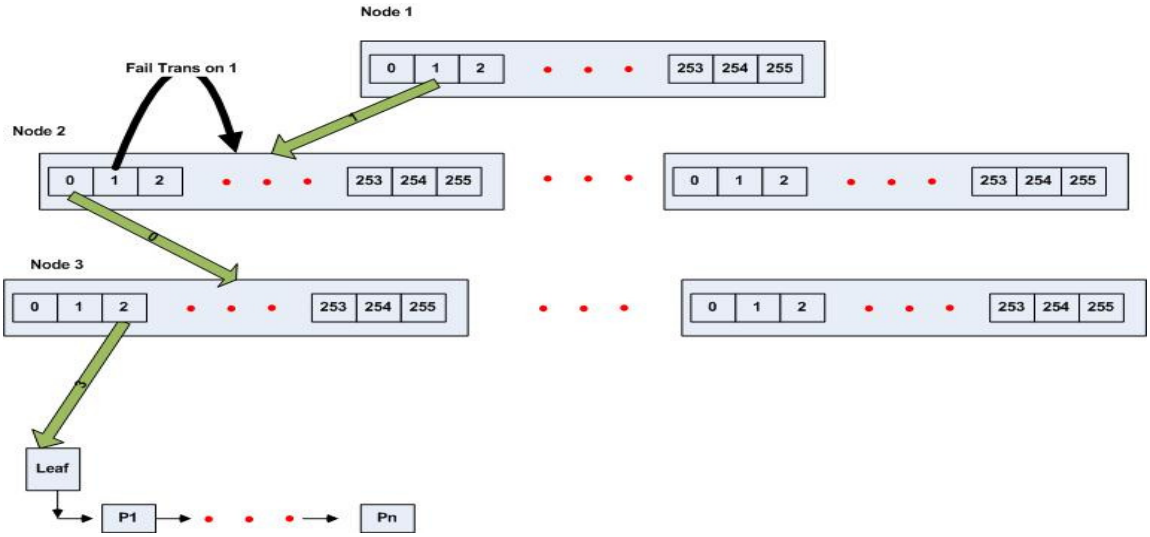


Figure 4.2: **ClamAV Aho-Corasick trie structure with arbitrary success transitions and one fail transition. Each transition represents an ASCII character. The maximum depth is 3. Patterns are added to linked lists after bypassing the maximum depth.**

ClamAV uses an array of linked lists to distribute the signatures among. To determine a location for a signature in the array, ClamAV hashes 3 characters of the signature using a hashing function. ClamAV tries to evenly distribute signatures in the array by trying the next 3-character sequence of the signature if the first 3 collides, and so on.

4.2.3 ClamWin and Clam Sentinel

ClamWin is a free antivirus for Microsoft Windows and used by more than 600,000 users worldwide on a daily basis [3]. It is based on the ClamAV scanning engine. ClamWin only supports on-demand scanning. Clam Sentinel is a free program that works with ClamWin to support on-access scanning.

4.3 Experimental methodology

Our experimental methodology was designed to answer the following two questions:

Question #1: Is there an exploitable timing channel based on how new signatures are added to ClamAV database? The basic idea behind the timing channel we demonstrate is to make the scanning engine hit in the place in which a signature is added over and over again to add a measurable delay to the scan. If the signature is there, then more work will take place and this means more scanning time. Three things are involved to make this happen. **First**, for any input to be scanned against database signatures, it needs to pass the filtering step. We extracted the exact 8 characters from each signature that are added to the filter, see Section 4.2.2 for more details about how the filtering works. When scanning files, ClamAV divides files into buffers of length 128KB. We need to make sure to plant the extracted characters in a buffer size basis rather than a file size basis. **Second**, for signatures that are added to the Boyer-Moore linked list, we extracted the characters from each Boyer-Moore signature that is used in the hashing function to add the signature to the linked lists, see Section 4.2.2 for more details about how the Boyer-Moore algorithm works. **Third**, for signatures that are added to the Aho-Corasick trie structure, we extracted the characters from each Aho-Corasick signature that will cause the scanner to go all the way down to the leaf nodes. See Section 4.2.2 for more details about how the Aho-Corasick algorithm works.

To demonstrate a possible timing channel, we collected the names of all of the viruses added since the first available ClamAV release, which was 17 April 2004. ClamAV maintains a virus database mailing list through which it reports virus addition or update. We downloaded and parsed all the HTML files since that date and put the results in a name-date list. Then, we unpacked the ClamAV database and removed from it all of the signatures of viruses that have their names on our list. This step makes the database as if it is the database from 17 April 2004. We made two kinds of experiments: the day-basis

experiment and the signature-basis experiment. For the day-basis experiment, we wrote a script that creates files per date/day. For example, if in date **D S** signatures are added to the database, then the script will create the Directory **D**. Inside **D** the script creates **S** directories each one corresponds to a virus signature. In each directory of **S** directories, the script creates 10 files each of which is 1MB. See Figure 4.3. The number and size of the files are below the default ClamAV limits to scan files. The 10 files' contents are identical. To create a file, we concatenate the extracted Boyer-Moore or Aho-Corasick characters from the corresponding signature until it reaches 1MB size. Meanwhile, we plant the filter characters of the signature every time we reach the buffer size. The initial content of the file depends on the file type a signature is taken from, see Section 4.2.2. The test begins by scanning the oldest date directory we have, then we add the signatures of that date to the database and scan it again. Then we scan the next date directory before and after adding that date's signatures, and so on. For the signature-basis experiment, we tested to see the effect of adding single signatures rather than the signatures for the whole day. For this experiment, ClamAV is asked to scan the 10 files, which consist of a signature's extracted characters, before and after adding that signature to the database.

Question #2: If the first question is confirmed, is it possible to exploit the timing channel in a real attack? Figure 4.4 illustrates a real-world scenario. A victim, who has ClamWin antivirus and Clam Sentinel installed, connects, through Internet Explorer, to a web server which is controlled by the attacker. Once connected, the victim is asked to download an ActiveX component that looks necessary to accomplish a certain task. Once downloaded, the ActiveX is started by JavaScript code. Then the ActiveX component creates a file that will be scanned by the antivirus and measures the CPU usage for a certain amount of time to determine the busy period the CPU experienced. We used the PDH (Performance Data Helper) library to query the processor time performance counter. This experiment starts when the CPU is almost idle. To determine the busy period of the CPU, we need a way to make sure that the CPU is busy with the antivirus rather than any other process which might be woken up or started at some time and then compute for a

small amount of time. So, the attacker can repeat the process to separate signal from noise coming from the CPU running other processes. What distinguishes the antivirus scanning process is that it keeps the CPU busy until it is done. This feature makes differentiating the start and the end time of the busy period easier. We determine the busy period in two stages: in the first stage, we start collecting all CPU usage samples (1 data point every 15 milliseconds) for 35 seconds, which is more than enough for the antivirus to scan a file, starting right after we close the file (the on-access scanner starts scanning right after closing the file). In the second stage, we take the averages of every contiguous 10-data-point sequence of the collected data from the first stage. We take the higher-than-normal averages (a threshold is set empirically) as start and end points and compute the elapsed scanning time based on this difference. After getting the total time the antivirus spent scanning the file, the ActiveX component triggers an event that will be received by a JavaScript function which, in turn, will send the results to the remote attacker.

A simpler model could be an insider threat which does not need a connection to a remote server.

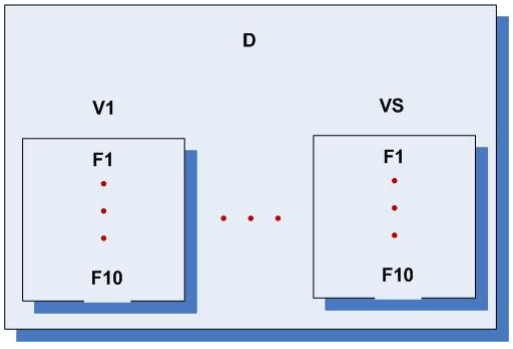


Figure 4.3: **Test file hierarchy per date D. The higher level directory D contains the files created for all signatures which were released on that day. Each V directory contains files created for only one signature.**

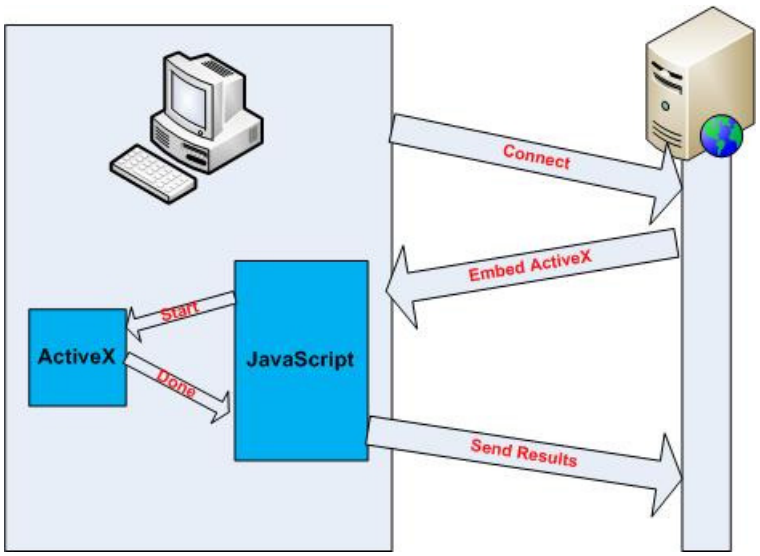


Figure 4.4: **A scenario for a real-world attack. The client uses Internet Explorer to connect to the server. Then, the client is asked to download an ActiveX component which a JavaScript script can control. The component creates a file and returns the CPU busy period, which will be considered as the scanning time, to the JavaScript as an event. The JavaScript sends the result back to the server.**

4.4 Results

In this section, we present results to answer the two questions we asked in Section 4.3. The results show that an attacker can exploit a timing channel from newly added signatures and that this attack can be implemented in a real-world scenario. We ran the first two experiments in Linux 2.6, on an Intel Core 2 Quad CPU at 2.66 GHz, and 8 GB RAM. Also, we ran the ActiveX experiments in Windows 7 OS, on an Intel Dual Core Atom CPU at 1.66 GHz, with 4 GB RAM.

4.4.1 Day-by-day experiment

In this experiment, we scanned a number of files that were created for each day, before and after adding the signatures of the day. See Section 4.3 for more details about the

setup. The **clamscan** command line program was used to initiate the scan. The averages were taken over 10 runs for each day. We take the differences between the scanning time averages after and before adding signatures (*i.e.*, the time to scan files after adding the signatures minus the time to scan files before adding the signatures).

Figure 4.5 shows a histogram of the differences represented by the ranges. For an x-axis value v, the corresponding y-axis value w is the number of occurrences that are in this range: ($v - 0.025$: v]. As expected, there are only 10 instances where the scanning times before and after adding signatures are almost the same. For all other instances, adding new signatures creates timing differences.

4.4.2 Single signatures experiment

In this experiment, we scan a number of files that are created to exploit only one signature each time. The **clamscan** command line program was used to initiate the scan. The averages were taken over 4 runs for each experiment. We take the differences between the scanning time averages after and before adding a signature. Figure 4.6 shows that a timing channel for one signature can be determined with high probability. For an x-axis value v, the corresponding y-axis value is the number of occurrences that are in this range: (v - 0.025: v].

4.4.3 ActiveX experiment for GENERIC type files

In this experiment, the ActiveX component creates 5 MB files and measures the time that the antivirus (ClamWin and Clam Centinal) spends on the CPU while scanning the files. Each run represents the creation and measurement time for only one file. This file is created by concatenating a randomly generated benign sequence of characters (*i.e.*, sequence of characters which has no effect in how a signature is inserted into Aho-Corasick tries or

Boyer-Moore linked lists) or by concatenating a randomly chosen extracted sequence of characters (*i.e.*, sequence of characters which affects the shape of the Aho-Corasick tries or Boyer-Moore linked lists). The files considered are of type GENERIC. Each run is repeated 5 times to ensure that the results are consistent. Figure 4.7 presents the averages over 5 runs for each file for both benign and extracted cases. Figure 4.8 zooms out on the area where both get closer. The results show that the scanning time for the extracted characters are always higher than that of the benign characters except for one data point (data point 2 in the case of extracted and 9 in case of benign), where both are equal.

In Figure 4.9, we present the worst case scenario where we compare the minimum runs in the case of the extracted characters to the maximum runs in the case of benign files. The results still show that the scanning time for the extracted characters are always higher than that of the benign ones except for two data points (data points 2 and 3 in the case of extracted and 9 in the case of benign). Figure 4.10 zooms out on the area where both get closer.

4.4.4 ActiveX experiment for HTML type files

In this experiment, instead of starting by filling characters into an empty file, we append characters to a basic HTML file. This would make the numbers different from the above experiment because the scanning engine will be directed to scan the files against signatures in the HTML (rather than GENERIC) root, see Section 4.2.2 for more information. Figure 4.11 shows the averages over 5 runs for each created file while Figure 4.12 represents the worst case scenario where we compare the minimum runs in the case of the extracted characters to the maximum runs in the case of the benign files.

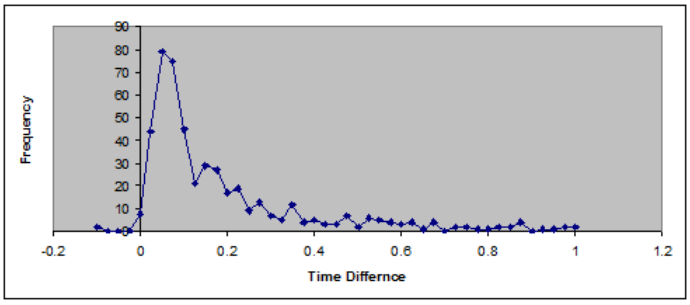


Figure 4.5: **Scanning time differences before and after adding signatures of a day.**

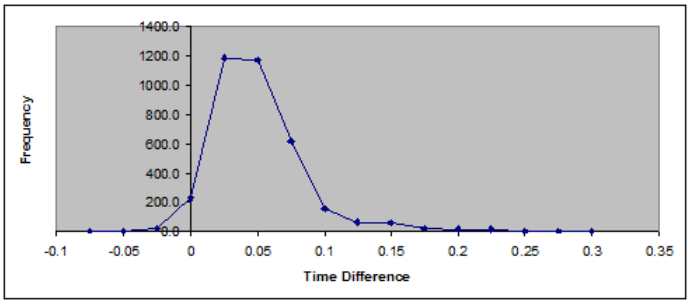
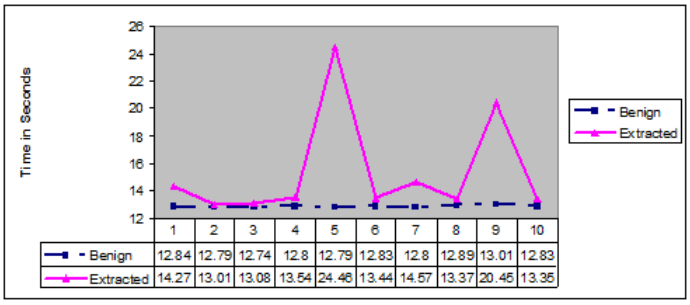


Figure 4.6: **Scanning time differences before and after adding a single signature.**

4.5 Discussion and future work

We argue that stealthily scanning clients is powerful because, besides being stealthy, it allows the attacker to gain a higher level of information. The notion of scanning clients, rather than servers, is a more fitting threat model for today's exploits that are based on "drive-by downloads." The user makes the initial connection and thus gives the attacker an opportunity to scan their machine. Web browsers are well known to be one of the main avenues for modern attack.

The running example of checking, through a timing channel, how up-to-date ClamAV is shows that application-level reconnaissance attacks are practical and can reveal high-level information about a user's system. The same general approach could be used to detect which antivirus program is installed (if any), the presence of other security software



	1	2	3	4	5	6	7	8	9	10
Benign	12.84	12.79	12.74	12.8	12.79	12.83	12.8	12.89	13.01	12.83
Extracted	14.27	13.01	13.08	13.54	24.46	13.44	14.57	13.37	20.45	13.35

Figure 4.7: **Scanning time of creating GENERIC type files out of benign and extracted characters. Each data point represents an average over 5 runs**

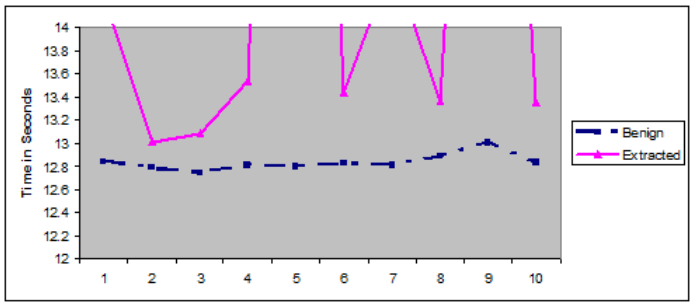
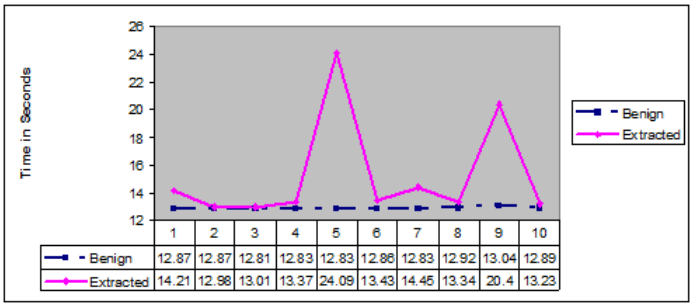


Figure 4.8: **The same experiment as in Figure 4.7 with clear border between benign and extracted.**

such as local intrusion detection systems or personal firewalls, if other malicious code is installed that hooks into some system behavior, mouse activity, the patch-level of the operating system, and more. Because all of these can be determined through timing channels, the attacker has a high degree of flexibility in the APIs used for scanning and need not have control of the system before doing reconnaissance.

Although our prototype attack, which was built for testing purposes to determine the time scales involved, is based on ActiveX the only three capabilities needed for a real application-level reconnaissance attack are to create or modify a file, measure the CPU usage, and keep track of time. Creating or modifying a file is possible even without APIs, simply by causing a file to be cached or a string of data to be logged. Furthermore, modern



	1	2	3	4	5	6	7	8	9	10
Benign	12.87	12.87	12.81	12.83	12.83	12.86	12.83	12.92	13.04	12.89
Extracted	14.21	12.98	13.01	13.37	24.09	13.43	14.45	13.34	20.4	13.23

Figure 4.9: **The same experiment as in Figure 4.7, but we show the worst case scenario.**

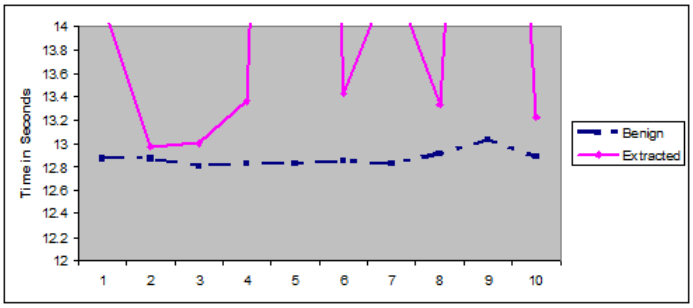


Figure 4.10: **The same experiment as in Figure 4.9, but we show the worst case scenario with clear border between benign and extracted.**

antivirus programs scan much more than just files and hook into a browser in many more places, so the surface available to scan them is larger than that of ClamAV. Measuring CPU usage can often be accomplished with several different APIs, or the attacker can simply measure the progress of one or more threads, which can be low priority need not necessarily consume all cores of the CPU if their timing for giving up the CPU is carefully orchestrated. Finally, keeping track of time is possible with a large variety of APIs, and need not be done on the client being scanned since the attacker's server can simply view events from the client and measure time itself. Crosby *et al.* [37] demonstrate that timing attacks can be performed over the internet with an accuracy of 15-100 microseconds, so in some cases it is not even necessary to use the clients timekeeping API.

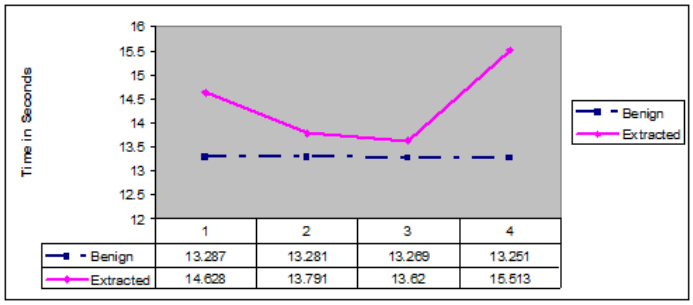


Figure 4.11: **Scanning time of creating HTML type files out of benign and extracted characters. Each data point represents an average over 5 runs.**

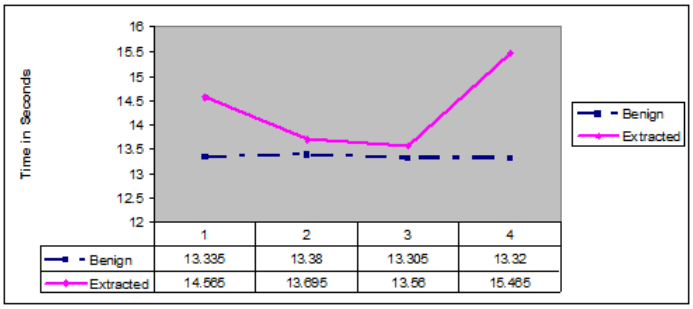


Figure 4.12: **The same experiment as in Figure 4.11, but we show the worst case scenario.**

We chose ClamAV for our tests because it is open source, the details of its core scanning algorithms are documented on the web, and the developer community for ClamAV is very helpful. We believe that more advanced antivirus programs, such as the proprietary antivirus programs that perform advanced filtering, emulation, algorithmic scanning, and heuristics, offer a lot more information that can be inferred from timing attacks than ClamAV. Table 4.1 shows how timing channel attacks could be possible in modern antivirus software. In general, the more sophisticated an antivirus program is, the more tradeoffs between performance and the number of patterns that can be detected must be traded off by amortizing fast codes paths *vs.* slow code paths. So, even though a closed-source, proprietary antivirus program would take a significant amount of more effort to develop

timing attacks for, the opportunities for gaining detailed high-level information about, for example, how up-to-date the signature database is will be much greater.

In terms of mitigation strategies to help ameliorate the type of high-level reconnaissance attacks we present in this chapter, strategies are needed that still allow for performance tradeoffs to be made for common cases. Predictive black-box mitigation [15] is a promising approach, and could potentially be applied by the antivirus program to maintain good performance while also minimizing the impact of timing channels attacks.

4.6 Related work

4.6.1 Network discovery

Network reconnaissance is very important first step in launching an attack. vPort scanning is a well known technique to probe networks and discover information about remote systems and the services they run. The threat model in port scanning is that the attacker, as a client, initiates the scanning victims who are servers that are reachable on the network. Nmap [64] is a well known tool to discover open, closed, and filtered ports, operating systems, services, and version information. All information that Nmap can get, however, is limited because it can only use raw packets and exposed services to discover information about networked hosts. Also, stateful and well configured firewalls and intrusion detection systems can stop many port scanning techniques. In our approach, the threat model is different. The attacker, who acts a server, waits for connections from victims, who are clients. Once connected, the attacker seeks to learn about the victim from the application layer, where a richer amount of information is available.

4.6.2 Timing channel attacks

Timing channel attacks are based on measuring the time it takes for a program to perform a task [17, 59, 100]. Timing channels have been exploited to reveal the secret keys in cryptographic systems [19, 84, 23], reveal SSH user passwords [90], breach users privacy [42], or detect virtual machines [47]. In this chapter, we exploited a timing channel attack in ClamAV antivirus by noticing the affect of adding a signature to the database on the scanning engine algorithm.

The work closest to our own is Bortz *et al.* [20]. Using timing attacks on web applications, they were able to find out private information about a user's web activity, such as the status and result of logins and login attempts, the number of objects on a page, and so forth. Our work extends timing attacks through the browser beyond web applications and shows that it is possible to find out security-relevant information about a potential victim machine's configuration.

4.6.3 Antivirus research

Attacking antivirus software is possible because antivirus is just software that could have vulnerabilities [1]. Because antivirus programs match data against a signature database, evading detection is possible using obfuscation transformations [29]. New signatures for obfuscated versions of viruses are generated based on samples of the newly obfuscated versions of that virus that are found in the wild. By measuring how up-to-date a potential victim's antivirus signature database is before attacking, an attacker can use older versions of their malicious code when possible and greatly reduce the exposure of their newest malicious codes.

Christodorescu *et al.* [30] shows that it is possible to extract the signature for a specific virus that the antivirus is using to detect that virus.

4.7 Conclusion

We showed that application-level reconnaissance through timing channels has the potential to reveal detailed, high-level information about a system to an attacker. The running example we used for the experimental results given in this chapter was based on checking how up-to-date the ClamAV antivirus on a given machine is. The results show that the attacker, with high accuracy, can determine if the database has been updated with certain signatures or not. Although most research concentrates on the potential of clients scanning servers, we concentrated in this chapter on the possibility of scans that a server might perform on a client. Also, the scans we considered occur at the application layer and can reveal much more information than, for example, port scanning. We believe that this type of reconnaissance will become increasingly important to study in the near future.

Technique	Description	Fast Scanning	Slow Scanning	Example
Algorithmic Scanning	Essential part of modern AV architecture. Implemented as java-like p-code (portable code) using a virtual machine	The input file does not trigger the algorithmic scanning	The file triggers the algorithmic scanning to run portable code (p-code) and its virtual machine which is hundreds of times slower than native machine code	When triggered, the algorithm to detect Zmist virus needs to execute at least 2 million p-code-based iterations [92]
Code Emulation	Powerful technique that emulates the execution through the CPU and the memory	The input file has no kind of encryption or suspicious patterns to trigger the emulation	Certain encrypted files can trigger the emulation	Fabi.9608 encrypted virus puts itself at the entry point of PE infected files. Although emulating the entry point of the infected file would expose the virus, it will significantly slow things down [92]
Heuristics	The structures of programs could trigger heuristics detection if they look suspicious	Programs' shapes look normal	Inconsistency between meta data and actual data in programs, or abnormality in the organization of program sections	Heuristic scanning is triggered when a PE program has an entry point pointing not to any of the sections but to an area after the header and before raw data (CIH-style viruses) [92]

Table 4.1: **Modern antivirus techniques still make timing channels possible.**

Chapter 5

Antivirus Usability: Antivirus Performance Characterization

It is well accepted that basic protection against common cyber threats is important, so it is recommended to have antivirus (AV) software installed on modern personal computers. However, what price do users pay in terms of performance, battery life, and other usability factors? Most effective AV products are commercial and thus for proprietary reasons the way that they work and how they affect the whole system is not openly published. Because it is important for security researchers and system developers to understand how exactly the AV impacts the whole system, in this work we take the approach of tracing operating system events. The AV adds layers of overhead and interfaces over systems that already have many complicated layers such as Microsoft Windows. Because there is very little open information in the literature to provide an understanding of how the AV's functionality impacts performance, our goal in this work is to shed some light on this.

To the best of our knowledge, this work is the first to present an OS-aware approach to analyse and reason about AV performance impact. Our results show that the main reason for performance degradation in the tasks we tested with AV software is that they mainly

spend the extra time waiting on events. Sometimes AV does cause some CPU overhead, but events such as hard page faults (*i.e.*, those that require disk accesses) are the main contributing factor to AV overhead. Because of the AV's intrusive behavior, the tasks in our experiments are caused to create more file I/O operations, page faults, system calls, and threads than they normally do without AV installed.

5.1 Introduction

Antivirus (AV) software, though vulnerable to many new attacks and unknown threats, is still widely used and recommended because it can detect a wide range of known malware threats. AV can also detect some unknown malware through heuristic algorithms [41] that it runs looking for malware-similar behaviors. According to two studies [4, 54], 80-81% of end-users have AV software installed on their computers. Also, Microsoft Windows alerts users who do not have AV installed and encourages them to install some form of AV.

Most AVs are commercial and their scanning algorithms are not revealed to the public. Even though this strategy is good for the AV venders to compete, it is not good for security researchers who will not be able to assess the AV's effects on the system. There is no effort in the security research literature that specifically targets analyzing commercial AV software. Understanding the ways that AV adds operating system events and layers of interfaces on top of existing systems is important not just for performance reasons, but also because the extra events have reliability and security implications. The delays caused by AV may exacerbate existing race conditions, for example. Also, viruses may use AV's performance overhead to fingerprint machines and gain information about what AV software is installed and how up-to-date it is [13]. Lastly, usability is a growing concern for security tools, yet AV impacts on usability are not well understood.

AV performance impact has not been well studied. Some studies [94, 8, 5] have con-

ducted experiments aiming to show the overhead (extra time or instructions) the AV adds while performing specific tasks. These studies did well in identifying AV performance overhead, but a major question still remains which is to know what exactly causes this overhead. In this work we consider the AV as a property of the whole system, and we use a system-wide inspection approach to characterize its performance based on different types of OS events. Although monitoring the system from the hardware-level [94] is useful and the hardware performance counters can give cycle-accurate information, the hardware view is limited in terms of information it can provide because of its low level of abstraction. Our approach is to have a system-wide, OS-aware instrumentation scheme that is able to provide information at different levels of abstraction.

In this work, we examine the performance issues caused by the AV from the OS point of view. We utilized an instrumentation tool to inspect the whole system: a Windows built-in technology, called Event Tracing for Windows (ETW). This technology is integrated with the Microsoft Windows kernel to log events of interest very efficiently, when enabled. More details about this technology are in Section 5.2. We designed several experiments that represent common end-user tasks with and without AV software installed to see how intrusive the AV is to these tasks and thus to pinpoint its performance impact. For more general results, we pursued two AVs: Symantec and Sophos.

5.2 Event Tracing for Windows (ETW)

Because the AV intercepts and inspects system-wide operations, we need a system-wide tool to be able to understand the AV's behavior. Event Tracing for Windows (ETW) is a low-overhead, system-wide instrumentation technology that comes with Microsoft Windows, starting from Windows 2000. ETW is integrated into the kernel so that it can capture most kinds of OS events, including process-related, CPU-related, IO-related, and memory-related events. ETW can be enabled or disabled at any time without any need to restart

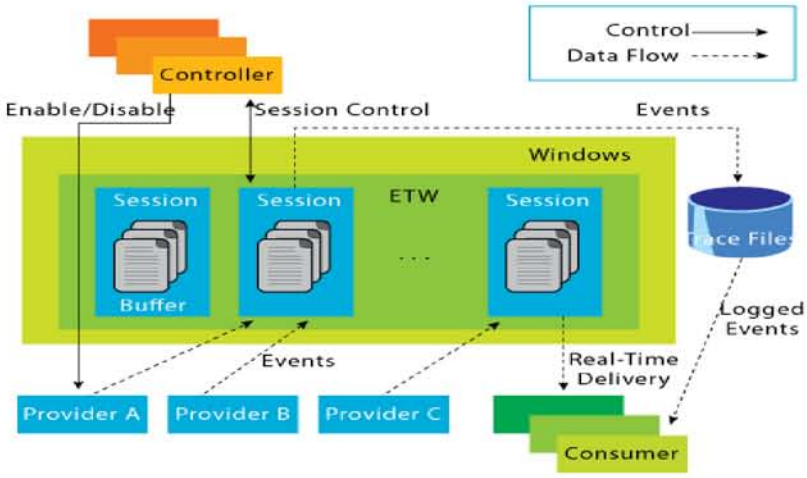


Figure 5.1: **ETW architecture, reproduced from [76].**

the machine. It produces binary files with a .etl extension that can be converted to CSV or XML formats using tools such as **tracerpt.exe**. In the XML format the trace file contains events, each starting with an <Event> tag and ending with a </Event> tag. Every event consists of a header and body. The header contains fixed and general information about the event and is common to all events, while the body contains specific information based on the event type. **Listing 1** shows an XML representation for Process Start event. The header starts with a <System> section, while the body starts with <EventData>.

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
    <System>
        <Provider Guid="{9e814aad-3204-11d2-9a82-006008a86939}" />
        <EventID>0</EventID>
        <Version>3</Version>
        <Level>0</Level>
        <Task>0</Task>
        <Opcode>1</Opcode>
        <Keywords>0x0</Keywords>
        <TimeCreated SystemTime="2011-03-21T21:37:15.770988400Z" />
```

```
        <Correlation ActivityID="{00000000-0000-0000-0000-000000000000}" />
        <Execution ProcessID="2120" ThreadID="6592" ProcessorID="0" KernelTime="0" ↩
            UserTime="15" />
        <Channel />
        <Computer />
    </System>
    <EventData>
        <Data Name="UniqueProcessKey">0xFFFFFA800E071060</Data>
        <Data Name="ProcessId">0x77C</Data>
        <Data Name="ParentId">0x848</Data>
        <Data Name="SessionId">       1</Data>
        <Data Name="ExitStatus">259</Data>
        <Data Name="DirectoryTableBase">0x14E594000</Data>
        <Data Name="UserSID">\\alsaleh-hpdv6\alsaleh</Data>
        <Data Name="ImageFileName">calc.exe</Data>
        <Data Name="CommandLine">&amp;quot;C:\Windows\system32\calc.exe&amp;quot; </Data>
    </EventData>
    <RenderingInfo Culture="en-US">
        <Opcode>Start</Opcode>
        <Provider>MSNT_SystemTrace</Provider>
        <EventName xmlns="http://schemas.microsoft.com/win/2004/08/events/trace">↩
            Process</EventName>
    </RenderingInfo>
    <ExtendedTracingInfo xmlns="http://schemas.microsoft.com/win/2004/08/events/trace">
        <EventGuid>{3d6fa8d0-fe05-11d0-9dda-00c04fd7ba7c}</EventGuid>
    </ExtendedTracingInfo>
</Event>
```

Listing 1

The ETW architecture consists of four components: controllers, providers, sessions, and consumers. See Figure 5.1.

1. **Controller :** the controller's main job is to start/stop tracing.

2. **Provider :** a provider is a source of events. Whenever an event happens, it sends it out to one of the sessions.

3. **Session :** sessions manage buffers and log events into trace files.

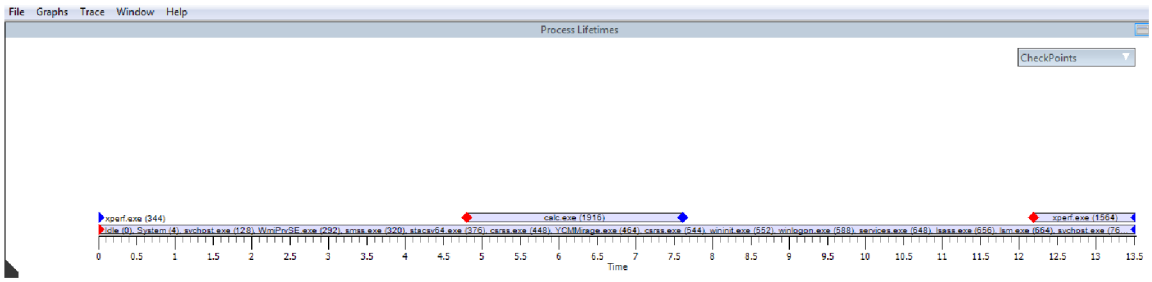


Figure 5.2: **xperf as a consumer.**

4. **Consumer :** consumers interpret the trace files produced by sessions.

The **xperf** tool comes with the Windows Performance Tools, and can be used as a controller to start ETW. It also can be used as consumer of the log files. Figure 5.2 shows a graph captured from xperf as a consumer.

The "Windows Kernel Trace" provider is responsible for sending the OS kernel events to the "NT Kernel Logger" session which in turn logs the events into trace files. In this project, we used xperf to enable The "Windows Kernel Trace" with many flags that represent all kinds of events.

5.3 Experimental setup

We designed our experiments to investigate and characterize the performance issues caused by AVs on specific tasks from a system-wide view. We designed four experiments that represent common end-user tasks. All of our experiments were scripted in PowerShell to avoid the need for any user intervention during the experiments. We ran the experiments on a machine that has Windows 7, an Intel Dual Core Atom processor at 1.66 GHz, 4 GB RAM, and 250 GB of hard disk space. The hard disk is partitioned into three partitions of roughly the same size. The machine has triple boot to boot into one of three Windows

7 images. All partitions have the same exact software except that the second partition has Sophos AV installed and the third partition has Symantec AV installed. Besides Windows 7, each partition has Microsoft Office 2010, Windows SDK 7.1 (which contains Windows Performance Tools 4.7), and Python. The Windows update service and the indexing services were disabled to prevent accidental events from taking place in the middle of the experiments. Also, we disabled the caching property of Symantec and purged Sophos' cache after each run to make sure that they are running from the same state in each experiment. We ran each experiment 10 times and computed the mean and the standard deviation to provide 95% confidence intervals. We compared the Vanilla case with the AV by conducting t-tests. A t-test measures if the means of two distributions are significantly different. The t-test score gives the probability of having observed the observed data if the means were not different. The smaller the t-test score, the more significant the difference. A p-score is 1 minus the t-test score, so a high p-score indicates a high confidence that the observed data shows a significant difference. We used a one-tailed, unpaired t-test that does not assume equal variances. All of the experiments in the following section were conducted on every partition separately and the machine was rebooted after every experiment before starting a new experiment.

5.3.1 Experiments

1. **Client-Server**: a PowerShell script starts the ETW logging, and then starts a Python client that connects to a server on another machine using sockets. The client then receives a compressed file from the server that is password-protected. The compressed file has putty.exe, the popular SSH client. After receiving the file, putty.a, the script uncompresses the file using 7za.exe, a decompression program. Once the decompression is done, the logging is disabled and the log file is taken. This experiment involves file I/O operations, system calls, CPU operations, and networking.

2. **Write to Microsoft Word and save**: a PowerShell script starts the ETW logging, creates a new Microsoft Word COM object, writes a short sentence to the object, saves and closes the object into a Word document call word.doc, and then stops logging. In this experiment we want to see how the AV interacts with such common operations that end-users frequently do. Memory and file I/O operations are involved in this experiment.

3. **Copy from Microsoft Word to Microsoft PowerPoint**: a PowerShell script starts the ETW logging, creates a new Microsoft Word COM object, opens a pre-created Word file (wordsourcedoc.doc) into the new object, copies its contents into the clipboard (the contents are only a short sentence), creates a new Microsoft PowerPoint COM object, creates a new presentation in the PowerPoint object, adds a new slide to the created presentation, pastes the copied sentence into the slide, saves the PowerPoint object, closes both objects, and stops logging. This experiment shows a frequent act in which end-users copy data from one application into another. The question is how the AV interacts with this process that involves COM object creation and data transfer between different applications.

4. **YouTube**: a PowerShell script starts the ETW logging and creates an Internet Explorer COM object and causes it to navigate to a particular video on www.youtube.com. Then the script sleeps for five seconds, letting the browser start the video, and then it closes the browser and stops logging. This experiment involves using a web browser, Internet Explorer, which is considered one of main avenues of malware threats to enter a system. Also, the experiment involves running the Adobe Flash plugin to stream the video over the internet.

For each experiment, we want to collect a variety of OS events that happen during the experiment. Afterwards, we want to see how the AV affects a task's performance and behavior by comparing the Vanilla and the AV cases and comparing the events. We want to see if the AV causes performance overhead to tasks, and then, if such the overhead exists,

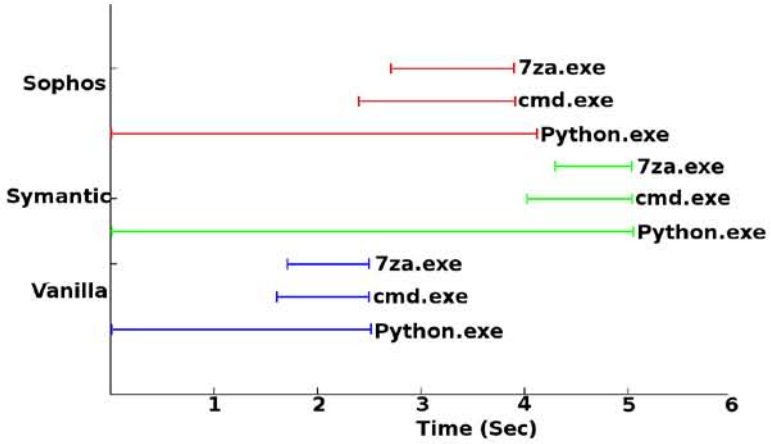


Figure 5.3: **Client-Server experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The processes we consider are: python.exe, cmd.exe, and 7za.exe.**

determine where this overhead is being spent by examining the task CPU scheduling. Then, we want to investigate the things that play main roles in deciding a task's CPU scheduling such as page faults, system calls, and file I/O operations. This gives a better reasoning approach for the AV performance overhead.

We convert the log files for all experiments into a CSV files and dump the data into a PostgreSQL [6] database. We designed SQL queries to retrieve relevant information. We present our findings in the next section.

5.4 Results

In this section we present the results for the experiments for which the methodology is explained in Section 5.3. The goal is to show as many differences (from the OS point of view) between running an experiment with and without an installed AV. In all of our experiments we consider only the processes and files directly involved in the experiment. The

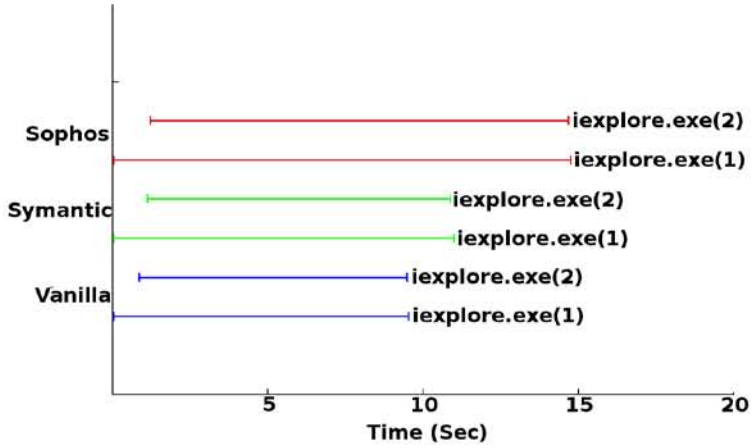


Figure 5.4: **YouTube experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The processes we consider are two processes of the same image: iexplore.exe.**

ETW framework makes it easy to isolate files and processes in this way. The OS metrics we examine are: file I/O operations, page faults, system calls, thread and process creation, and CPU scheduling. The error bars in all graphs represent 1.96 standard deviations to show a 95% confidence interval.

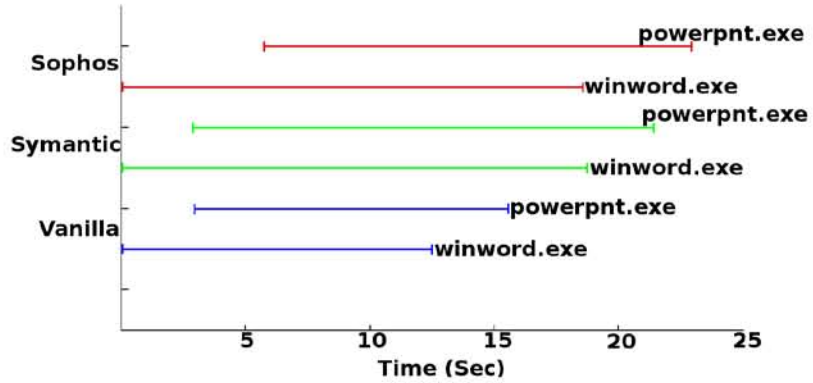


Figure 5.5: **Copy from Microsoft Word to Microsoft PowerPoint experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The processes we consider are: winword.exe and powerpnt.exe.**

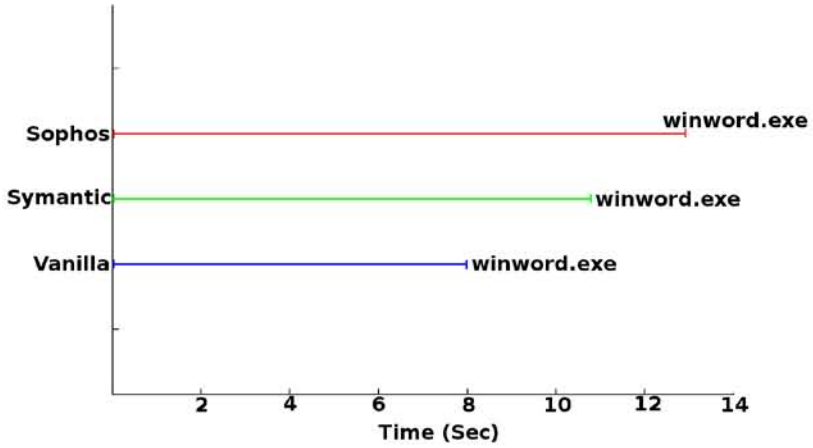


Figure 5.6: **Write to Microsoft Word experiment: processes' lifetimes. Each line represents the time a process takes from start to end. The process we consider is winword.exe.**

Figures 5.3, 5.4, 5.5, and 5.6 show the lifetimes of processes we considered for all experiments. The overhead caused by both Symantec and Sophos is apparent in all experiments. Furthermore, if a process start time depends on an event from another process, the other process can delay the start time of its dependent.

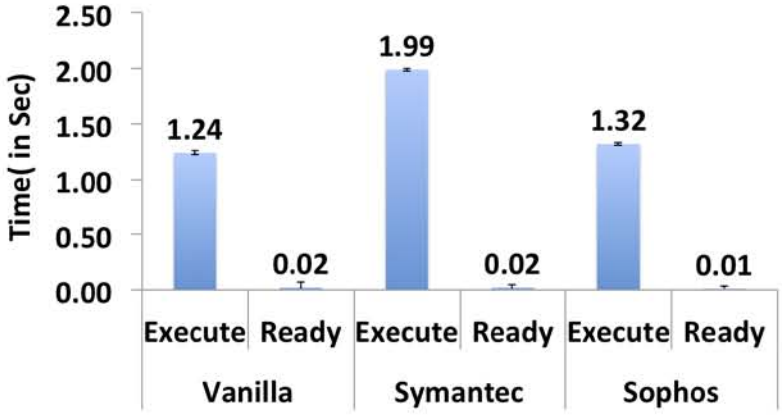


Figure 5.7: **Client-Server experiment: process average return time (in sec) distributed between execute and ready states.**

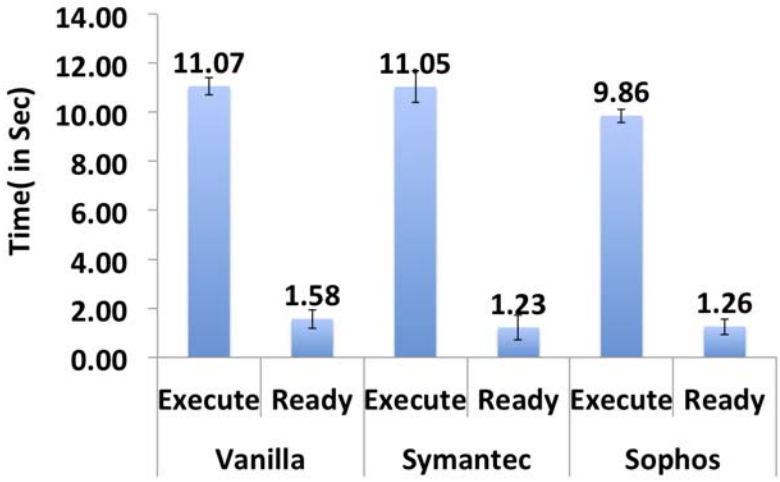


Figure 5.8: **Copy from Microsoft Word to Microsoft PowerPoint experiment: process average return time (in sec) distributed between execute and ready states.**

During its execution lifetime, a process is in one of three states: executing, waiting in the ready queue to be scheduled on a CPU by the scheduler, or waiting for an event that, when this event happens, puts the process back in the ready queue. Figures 5.7, 5.8, 5.9, and 5.10 show the time that processes spend in execute and ready states. It is apparent

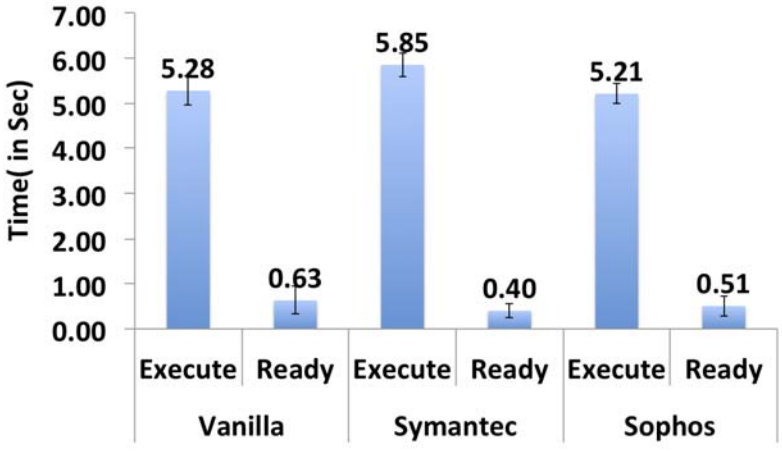


Figure 5.9: **Write to Microsoft Word experiment: process average return time (in sec) distributed between execute and ready states.**

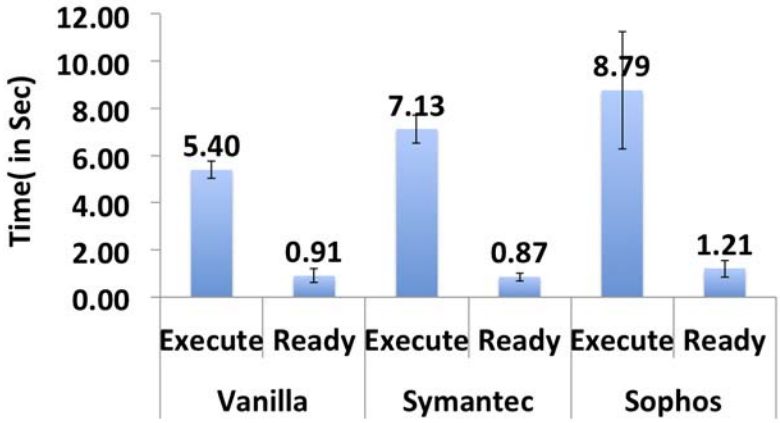


Figure 5.10: **YouTube experiment: process average return time (in sec) distributed between execute and ready states.**

from figures 5.7, 5.8, and 5.9 that the main reason for the overhead caused by the AV on the running processes comes from waiting on events because the times spent on execute and ready states for the Vanilla and AV cases are not different enough to explain AV overhead. Figure 5.10 shows that in case of Symantec the main reason for the overhead

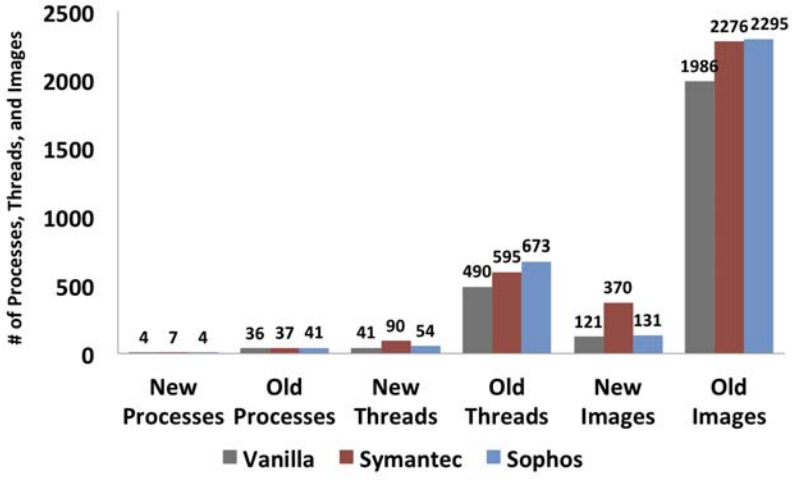


Figure 5.11: **Client-Server experiment: the total number of processes, threads, and images in the system before and during the experiment.**

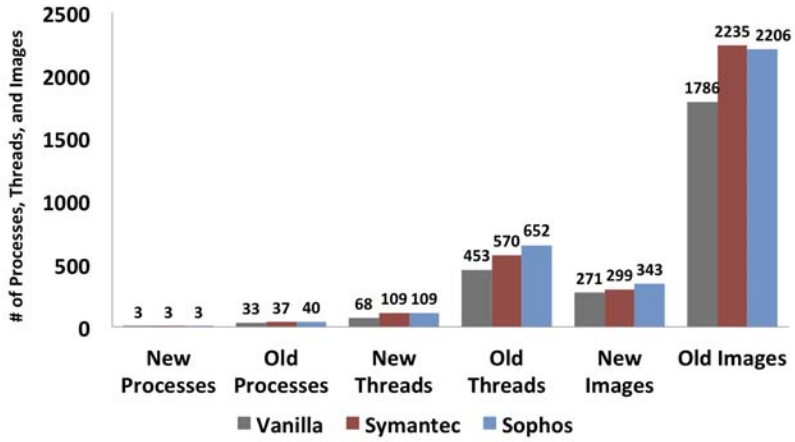


Figure 5.12: **YouTube experiment: the total number of processes, threads, and images in the system before and during the experiment.**

is the extra time that processes spend on using the CPU (see figure 5.4 for the overhead), while in the Sophos case the overhead is distributed between CPU and waiting on events.

A process' lifetime is affected by the system (hardware and software) it is running on. Because we have fixed the hardware for all experiments, it is only the software that causes variation in these experiments. Figure 5.11 shows the number of processes, threads, and images that were created/loaded before and during running the Client-Server experiment. The figure shows that the AV creates threads and loads images during the experiments to achieve some scanning task. Figure 5.12 shows the same happening in the YouTube experiment. Although this does not have a direct performance implication, it shows some intrusiveness from the AV on the tasks. Also, those extra threads could compete with the main tasks for some resources like the hard drive and memory.

All the t-test results for Figures 5.13 through 5.24 report more than 0.99 p-scores except for Figures 5.23 and 5.24 where the p-scores for Vanilla compared with Symantec are 0.73 and 0.90, respectively.

Figures 5.13, 5.14, 5.15 , and 5.16 show the numbers of file I/O operations on the files

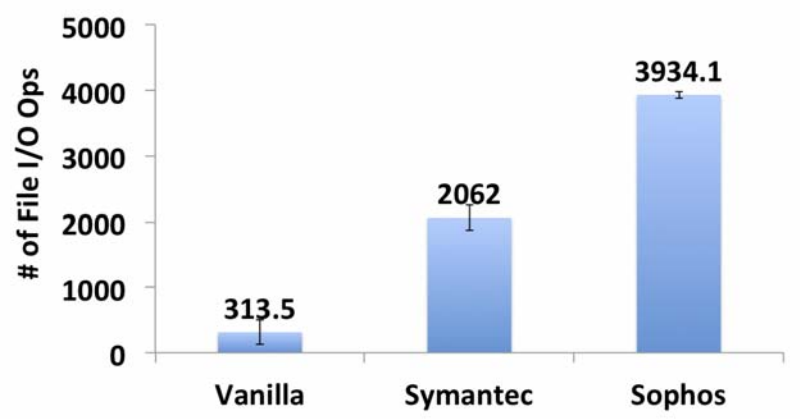


Figure 5.13: **Client-Server experiment: the average of the total number of file I/O operations with respect to the files involved in the experiment (python.exe, 7za.exe, cmd.exe, client.py, putty.a, putty.exe).**

involved in different experiments by whatever process, including the AV. The intrusiveness of both AVs is also apparent in these figures.

Figures 5.17 and 5.18 show the total number of file I/O operations directly made by the

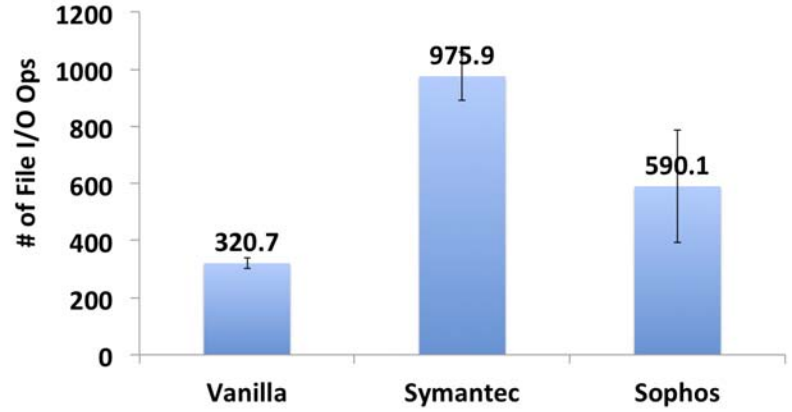


Figure 5.14: **YouTube experiment: the average of the total number of file I/O operations with respect to the files involved in the experiment (iexplore.exe). A one-tailed t-test for the difference between Vanilla and Sophos being at least 210 gives a p-score of 0.95, meaning that we can be confident that Sophos is at least 210 larger than Vanilla.**

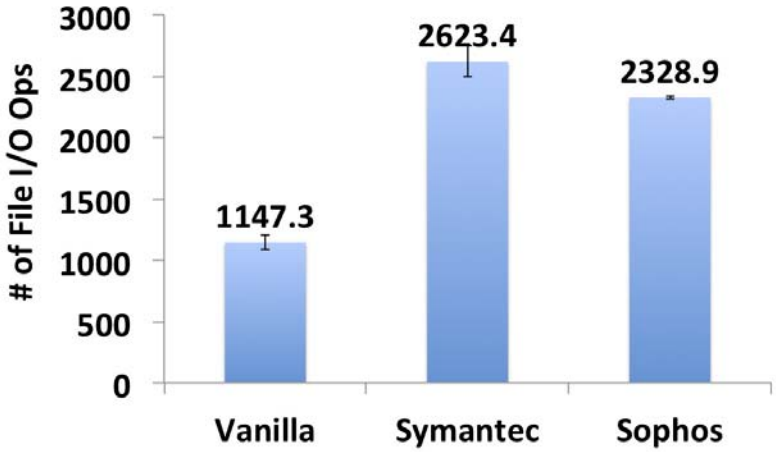


Figure 5.15: **Copy from Microsoft Word to Microsoft PowerPoint experiment: the average of the total number of file I/O operations with respect to the files involved in the experiment (winword.exe, powerpnt.exe, wordsourcedoc.doc, pres.ppt).**

processes we considered. Symantec forces some processes to make file I/O operations that they would not otherwise make. For example, python.exe performed 128 file I/O operations on a "ProgramData/Symantec/Definitions/VirusDefs/20110313.002/VIRSCAN7.DAT"

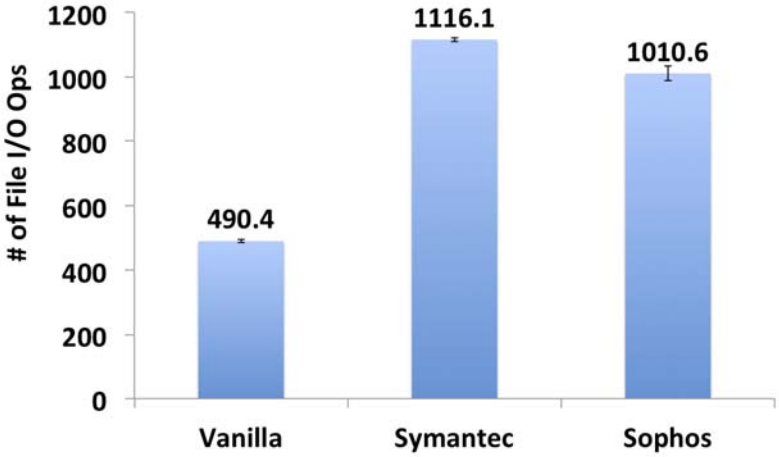


Figure 5.16: **Write to Microsoft Word experiment: the average of the total number of file I/O operations with respect to the files involved in the experiment (winword.exe, word.doc).**

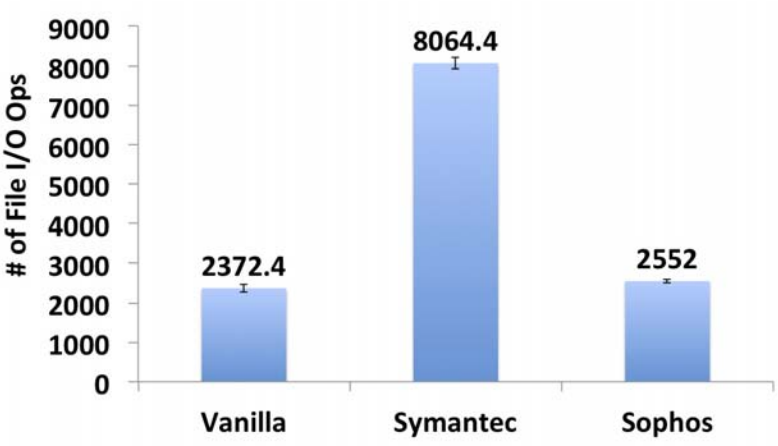


Figure 5.17: **Client-Server experiment: the average of the total number of file I/O operations invoked by the processes involved in the experiment. A one-tailed t-test for the difference between Vanilla and Sophos being at least 145 gives a p-score of 0.95, meaning that we can be confident that Sophos is at least 146 larger than Vanilla.**

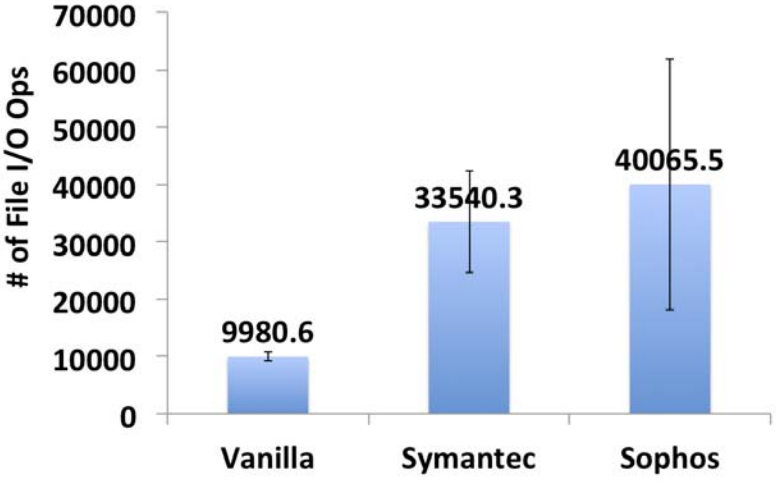


Figure 5.18: **YouTube experiment: the average of the total number of file I/O operations invoked by the processes involved in the experiment. A one-tailed t-test for the difference between Vanilla and Sophos being at least 23500 gives a p-score of 0.95, meaning that we can be confident that Sophos is at least 23500 larger than Vanilla.**

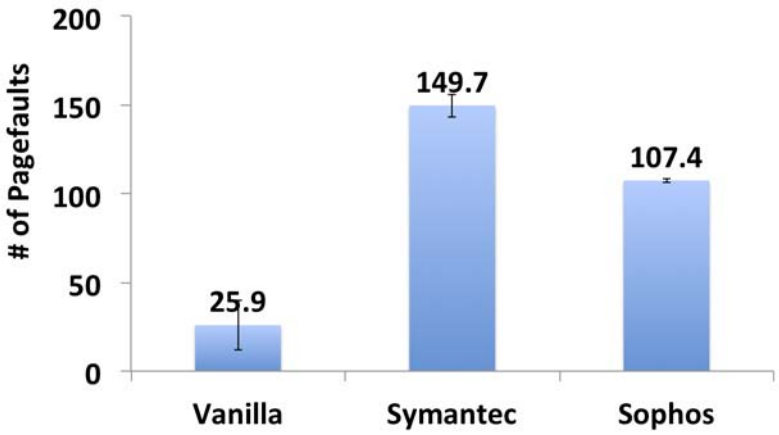


Figure 5.19: **Client-Server experiment: the average of the total number of page faults caused by the processes involved in the experiment.**

file. The python.exe process does not read Symantec's signatures during normal execution, *i.e.*, in the Vanilla case. Sophos, on the other hand, forces the processes to make file I/O operations on "sophos_detoured.dll, sophos∼1.dll", and "swi_lsp.dll" (Sophos Web Intelligence).

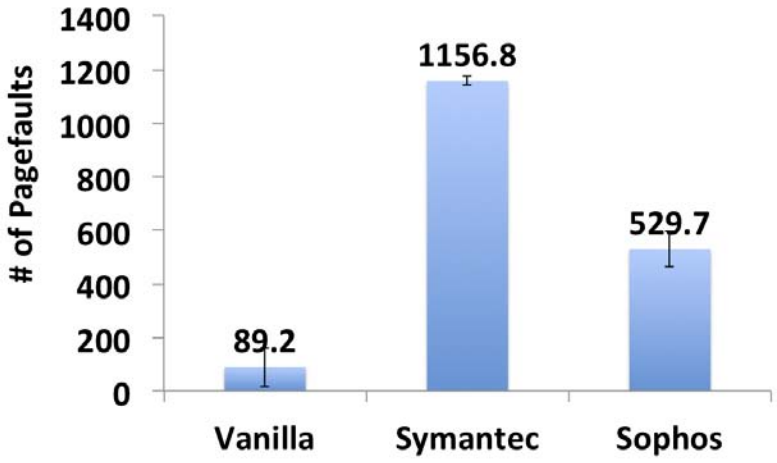


Figure 5.20: **Write to Microsoft Word experiment: the average of the total number of page faults caused by the processes involved in the experiment.**

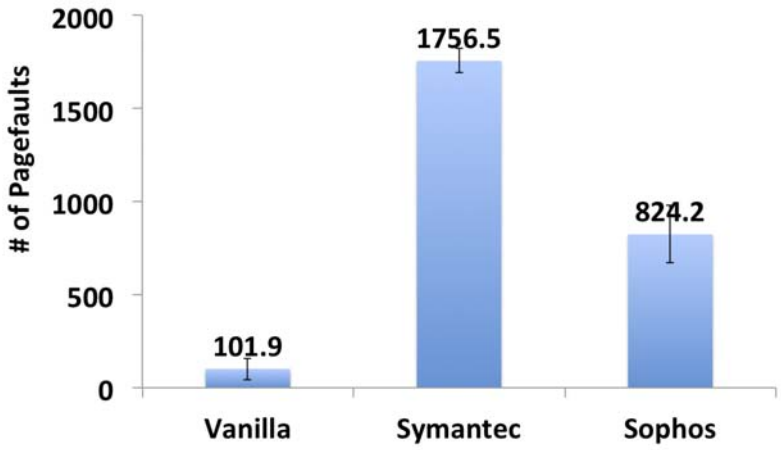


Figure 5.21: **Copy from Microsoft Word to Microsoft PowerPoint experiment: the average of the total number of page faults caused by the processes involved in the experiment.**

Figures 5.19, 5.20, and 5.21 show the number of caused page faults. When a process causes a hard fault, it needs to wait until the page is read into memory from the hard disk. The high increase of hard faults in both the Symantec and Sophos case experiments explains the overhead they add to the processes' lifetimes. Again, these faults are only

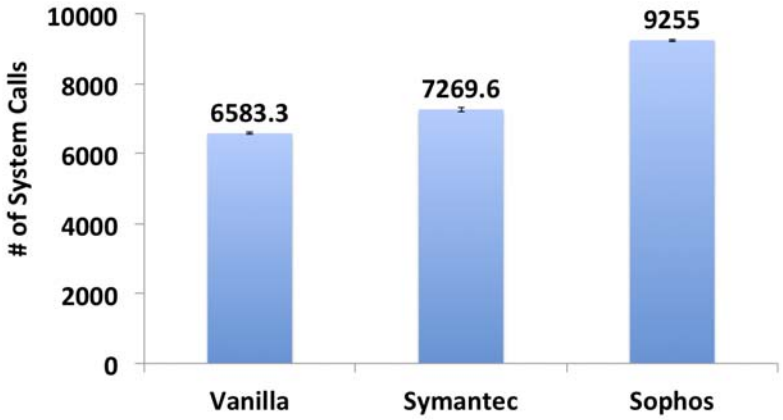


Figure 5.22: **Client-Server experiment: the average of the total number of system calls made by the processes involved in the experiment.**

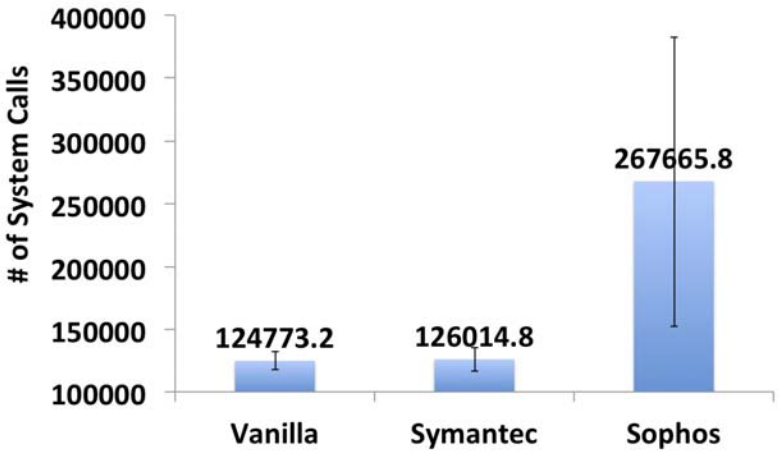


Figure 5.23: **YouTube experiment: the average of the total number of system calls made by the processes involved in the experiment. A one-tailed t-test for the difference between Vanilla and Sophos being at least 108,000 gives a p-score of 0.95, meaning that we can be confident that Sophos is at least 108,000 larger than Vanilla.**

accumulated for the processes we considered.

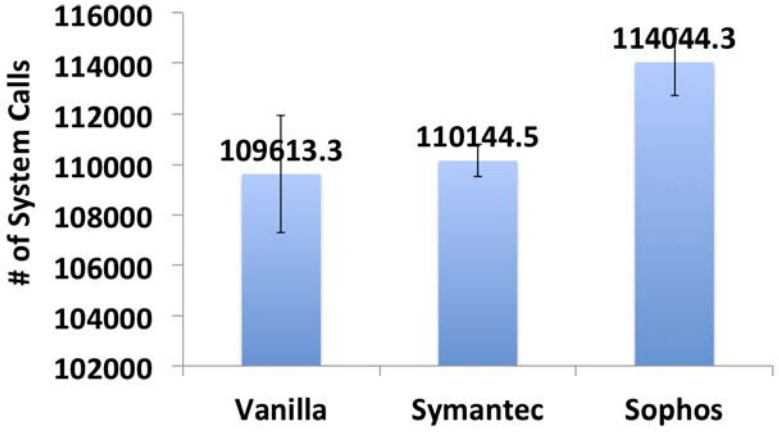


Figure 5.24: **Write to Microsoft Word experiment: the average of the total number of system calls made by the processes involved in the experiment. A one-tailed t-test for the difference between Vanilla and Sophos being at least 3670 gives a p-score of 0.95, meaning that we can be confident that Sophos is at least 3670 larger than Vanilla.**

A system call proceeds from the user space (unprotected mode) to kernel space (protected mode) for the kernel to achieve a task on behalf of the user process and then reschedule the user process when the kernel's task is done. Increasing the number of system calls decreases performance. Figure 5.22 shows the increase in the number of system calls made by the processes we care about in the cases of Symantec and Sophos, compared with the Vanilla case. Figures 5.23 and 5.24 show the increase in system calls in the case of Sophos, but in the case of Symantec the number of system calls is not significantly different from that of Vanilla.

5.5 Discussion and future work

Although we claim that our approach is useful in characterizing the effect of AV programs on running processes in terms of performance from the OS point of view, it is not perfectly accurate. We depend on finding differences between the Vanilla and the AV cases to make conclusions, which is not tightly coupled what the AV is actually doing. Some operations might happen or not happen based on the system state or the state of the task that is about to execute. For example, if an AV uses the same DLLs the task will be using, then the task would not cause a hard fault in case of the AV because the AV might have already caused the DLLs to have been brought in from the disk before the task starts. Also, a program could perform some operations based on its last run or configuration, which might make its execution performance profile look different. To deal with this problem, we repeated each experiment 10 times and computed the mean and 95% confidence interval for them.

In this work, we started from the files and processes we needed to consider because they were related to the task being tested, to find out what operations they perform or are performed on them. Another approach, which is a possible future work, is to start from the AV components and directly find out what exactly they are doing through a reverse-engineering process. This approach is challenging, though, because we would need to

know all of the changes and the components the AV adds to the system when it is installed.

5.6 Related work

The AV software is well known in the literature to prevent viruses and worms from spreading, including known threats and, through heuristics, some unknown threats. Surfing the internet without having an AV program installed is not considered safe. Although pattern matching is at the heart of every AV scanning engine, other techniques such as heuristics [41], code emulation, and algorithmic scanning are essential parts of modern AVs [92].

Little research has been conducted towards analyzing and improving AVs, mainly because most AVs are closed source. A few papers [61, 67, 77, 96, 7, 87, 49] have focused on improving the AV scanning time. Lin *et al.* [61] proposed a hybrid algorithm to enhance the scanning engine of ClamAV [2]. Miretskiy *et al.* [67] designed an on-access AV file system that accumulatively scans files under operations in ClamAV. Paul *et al.* [77] aimed at improving and speeding up the AV scanning engine by utilizing the disk drive processor. Vasiliadis *et al.* [96] also provided a parallel AV scanning engine by utilizing modern graphics processors (GPUs). Several other works [7, 87, 49] suggest using a specialized hardware for virus scanning. What is common to all of these previous works is that they try to enhance the AV scanning time by some means. In our work, we focus instead on exposing the direct effects of the AV on tasks from the OS point of view.

Al-Saleh *et al.* [13] shows that it is possible to create timing channel attacks against AVs that can reveal the presence of AV and how up-to-date the AV's' signature database is, even to the granularity of individual signatures, by sending carefully crafted strings of bytes to a remote machine that do not trigger AV but cause differences in response times. Christodorescu *et al.* [30] shows that it is possible to extract the signature for a specific virus that the AV is using to detect that virus. The closest to our work is Ulusji

et al. [94], however they studied the performance of the AV purely at the hardware level while our approach is to characterize the AV performance from the OS point of view. More limited ways of measuring AV performance [8, 5] are conducted by running a task and then comparing the total time while running different AVs, our study seeks to provide more details by explaining how the extra time is spent in terms of OS events such as page faults and system calls.

Software instrumentation [56, 22, 71, 25, 88] is a powerful technique to analyze programs' behaviors through adding extra code at certain positions of the instrumented programs. The purpose of instrumentation could be measuring performance, proving correctness, or assuring the security of programs. Pin [56] is a dynamic binary instrumentation tool that can instrument binaries at the instruction level without modifying them. DTrace [25] is a whole-system instrumentation tool for UNIX-based systems such as Solaris and Linux. ETW is similar to DTrace, but ETW is integrated within the Microsoft Windows kernel.

5.7 Conclusion

Less care has been given by the research community to study the functionality and performance impact of AV software despite how important and widely used it is. In this work, we investigated the performance issues for two commercial AVs: Symantec and Sophos. While previous techniques to studying AV concentrated on the elapsed times of running tasks with the existence of AVs or viewed the problem from a purely hardware point of view, we focused on an OS-aware approach to give more meaningful and accurate results that include the reasons for increased return times for tasks. We used the Event Tracing for Windows instrumentation technology (a system-wide, kernel-integrated tool), xperf (a performance tool), and PowerShell (an automation and scripting framework in Windows) to design our experiments so that the performance overhead experienced by real users

was represented in our experiments. Our experiments were designed to simulate common end-user tasks. Our results show that a considerable amount of performance overhead is added by the AVs because of the AV program's intrusiveness. The main impact of the AV on tasks is that they spend extra time waiting on events. The AV changes a task's behavior by forcing it do more file I/O operations and system calls, causing more page faults, and creating more threads. We also showed that a process' behavior is changed with the existence of an AV program. So, for software development processes (design, testing, debugging), intrusion detection systems (which might look at a process changing its behavior as anomalous), and other system-wide performance and behavior-based security issues, the AV is an important factor that should be taken into consideration in future efforts to design secure systems.

Chapter 6

Conclusion

Although the trade-off between usability and security is well known, efforts to understand these trade-offs and improve them should be continuous to achieve a better balance. Not only are security tools supposed to secure systems against attacks, but also they are required to not degrade the usability of systems by, for example, degrading the performance or providing very restricted environments which systems' users can not use effectively. Towards achieving better security and usability, in this dissertation we examined the internals of modern security tools and reasoned about security and usability problems which previous research had not identified because of limited experiments and controlled environments. This dissertation studies two modern security tools, Dynamic Information Flow Tracking (DIFT) and Antivirus (AV) software, for their importance and wide usage.

Even though applying all propagation rules in a DIFT system is admittedly important, some of those rules are dropped from current systems for usability (stability) reasons. Because we implemented and experimented with DIFT systems that applied all rules, we were able to identify the contributing factors to the DIFT overtagging problem and provide a practical approach to deal with it in two ways. First, to show that systems complication (software and hardware) contributes to the overtagging problem in DIFT, we successfully

applied DIFT to sensor network devices. DIFT for sensor network devices provides a secure and usable system that accommodates the special requirements of a sensor network device. Second, in our DIFT implementation for general purpose computers as an intrusion detection system, tags are the mutual information between sources and destinations. We also introduced the notion of lifetime usage for tags as a main strategy to deal with overtagging. Our work suggests that accounting for data sources and having context-aware tagging to avoid overtagging are important to achieve accurate dynamic information flow in any future work.

The second security tool this dissertation is interested in is the Antivirus (AV) software because it has received less attention from tje research community, despite its importance and wide usage. Besides the possibility of being evaded by new attacks that threaten AV security, the AV should be aware of timing channel attacks that the attacker could use to learn details about the AV and its signature database based on the time the AV spends on scanning carefully crafted inputs. By studying the ClamAV internals, we were able to launch a timing channel attack against it to infer how up to date ClamAV is, based on the scanning time. We also show that commercial AVs are susceptible to the same kinds of attacks. The possibility for the attacker to remotely fingerprint a client AV is a future work.

We also studied the AV performance impact and intrusiveness on systems, because these are main factors towards the AV usability. The AV installs system-wide hooks to inspect system operations. To study the AV effects on a system, we need to have a system-wide instrumentation tool that is able to account for all kinds of OS events. Event Tracing for Window (ETW) is such an instrumentation tool that does exactly that. By dumping the ETW data into PostgreSQL, we were able to aggregate information about what happens with and without the existence of the AV, and make comparisons. We showed that the AV affects tasks by causing them to spend more time waiting on events or using the CPU. The AV needs to be less intrusive to tasks for it to be completely acceptable and thus

usable by users, especially in constrained environments such as laptops. We expect this research to establish a foothold towards analyzing the AV software in detail rather than dealing with it as an unknown within a system. As a future research, we intend to study the AV performance impact by tracking the changes and the components the AV adds to the system when it is installed.

The thesis that this dissertation tested was the claim that we can reason about security and usability trade-offs in fine-grained ways by building and testing full systems. We have demonstrated this through testing various aspects of DIFT and AV.

References

[1] Attacking antivirus. http://www.blackhat.com/presentations/bh-europe-08/Feng-Xue/Whitepaper/bh-eu-08-xue-WP.pdf.

[2] Clam antivirus. http://www.clamav.net.

[3] Free antivirus for windows. http://www.clamwin.com.

[4] Internet security threats will affect U.S. consumers holiday shopping online. http://www.bsacybersafety.com/news/2005-Holiday-Online-Shopping.cfm.

[5] Passmark software. http://www.passmark.com/benchmark-reports/.

[6] Postgresql: The world's most advanced open source database. http://www.postgresql.org/.

[7] Tarari: Anti-virus content processor. http://www.tarari.com/antivirus/index.html.

[8] Tests of anti-virus and security software. http://www.av-test.org/.

[9] TinyOS. http://tinyos.net.

[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and Communications Security*, pages 340–353, New York, NY, USA, 2005. ACM.

[11] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and System Security*, 2006.

[12] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[13] M. I. Al-Saleh and J. R. Crandall. Application-level reconnaissance: Timing channel attacks against antivirus software. In *4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2011.

[14] anonymous. Once Upon a free()..., Phrack 57.

[15] A. Askarov, A. C. Myers, and D. Zhang. Predictive black-box mitigation of timing channels. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, New York, NY, USA, 2010. ACM.

[16] B. Babayan. Security, www.elbrus.ru/mcst/eng/ SECURE_INFORMATION_SYSTEM_V5_2e.pdf.

[17] H. Bar-El. Introduction to side channel attacks.

[18] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In EICAR, pages 180–192, 2006.

[19] D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[20] A. Bortz, D. Boneh, and P. Nandy. Exposing private information by timing web applications. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 621–628, New York, NY, USA, 2007. ACM.

[21] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[22] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2000.

[23] D. Brumley and D. Boneh. Remote timing attacks are practical. In *In Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[24] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*. ACM Press, Oct. 2008.

[25] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[26] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] S. Chen, J. Xu, and E. C. Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[28] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*, August 2004.

[29] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[30] M. Christodorescu and S. Jha. Testing malware detectors. *SIGSOFT Softw. Eng. Notes*, 29(4):34–44, 2004.

[31] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[32] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Can we contain internet worms? In *HotNets III*, 2005.

[33] J. R. Crandall and F. T. Chong. A Security Assessment of the Minos Architecture. In *Workshop on Architectural Support for Security and Anti-Virus*, Oct. 2004.

[34] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[35] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[36] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[37] S. A. Crosby, D. S. Wallach, and R. H. Riedi. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3):1–29, 2009.

[38] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing hardware architectures for security. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2006.

[39] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[40] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM.

[41] J. Eisner. Understanding Heuristics: Symantecs Bloodhound Technology. Symantec white paper series volume xxxiv. `http://www.symantec.com/avcenter/reference/heuristc.pdf`, 1997.

[42] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *CCS '00: Proceedings of the 7th ACM conference on Computer and Communications Security*, pages 25–32, New York, NY, USA, 2000. ACM.

[43] J. S. Fenton. Information protection systems. In *Ph.D. Thesis, University of Cambridge*, 1973.

[44] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[45] C. Ferguson, Q. Gu, and H. Shi. Self-healing control flow protection in sensor applications. In *WiSec '09: Proceedings of the second ACM conference on Wireless network security*, pages 213–224, New York, NY, USA, 2009. ACM.

[46] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA, 2008. ACM.

[47] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *SIGOPS Oper. Syst. Rev.*, 42(3):83–92, 2008.

[48] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, 2003.

[49] J. K. H. M. H. Gray and D. L. Wakelin. In transit detection of computer virus with safeguard. us patent 5319776.

[50] Q. Gu and R. Noorani. Towards self-propagate mal-packets in sensor networks. In *WiSec '08: Proceedings of the first ACM conference on Wireless network security*, pages 172–182, New York, NY, USA, 2008. ACM.

[51] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[52] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41, New York, NY, USA, 2006. ACM.

[53] IEEE 1149.1, JTAG: Standard Test Access Port and Boundary-Scan Architecture.

[54] M. Kaiser. Small and Medium Size Businesses are Vulnerable. 2009.

[55] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *DSN '09: Proceedings of the International Conference on Dependable Systems and Networks*, pages 115–124. IEEE Computer Society, 2009.

[56] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[57] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2007. ACM.

[58] R. Kumar, E. Kohler, and M. Srivastava. Harbor: software-based memory protection for sensor nodes. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349, New York, NY, USA, 2007. ACM.

[59] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[60] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[61] P.-C. Lin, Y.-D. Lin, and Y.-C. Lai. A hybrid algorithm of backward hashing and automaton tracking for virus scanning. *IEEE Transactions on Computers*, 60:594–601, 2011.

[62] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[63] S. B. Lipner. A comment on the confinement problem. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles*, pages 192–196, New York, NY, USA, 1975. ACM Press.

[64] G. F. Lyon. *Nmap Network Scanning*. Insecure.Com LLC, 2008.

[65] P. Malacaria. Assessing security threats of looping constructs. In *POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 2007. ACM Press.

[66] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 9–11, 2008.

[67] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium (Security 2004*, pages 73–88. USENIX Association, 2004.

[68] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[69] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. *SIGARCH Comput. Archit. News*, 36(1):211–221, 2008.

[70] Nergal. The advanced return-into-lib(c) exploits: PaX case study, Phrack 58.

[71] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Third Workshop on Runtime Verification (RV03)*, 2003.

[72] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 73–85, New York, NY, USA, 2009. ACM.

[73] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.

[74] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 308–318, New York, NY, USA, 2008. ACM.

[75] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[76] D. I. Park and R. Buch. Improve debugging and performance tuning with ETW. `http://msdn.microsoft.com/en-us/magazine/cc163437.aspx/`.

[77] N. R. Paul. *Disk-level behavioral malware detection*. PhD thesis, Charlottesville, VA, USA, 2008. Adviser-Evans, David.

[78] K. Piromsopa and R. J. Enbody. Defeating buffer-overflow prevention hardware. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2006.

[79] J. Polley, D. Blazakis, J. Mcgee, D. Rusk, and J. S. Baras. Atemu: A fine-grained sensor network simulator. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[80] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[81] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[82] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Efficient type and memory safety for tiny embedded systems. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems*, page 6, New York, NY, USA, 2006. ACM.

[83] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems, 1975.

[84] W. Schindler. A timing attack against rsa with the chinese remainder theorem. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 109–124, London, UK, 2000. Springer-Verlag.

[85] scut. Exploiting Format String Vulnerabilities.

[86] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In S. De Capitani di Vimercati and P. Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, Oct. 2007.

[87] M. Silberstein. Designing a cam-based coprocessor for boosting performance of antivirus software, 2004.

[88] A. Skaletsky, T. Devor, N. Chachmon, R. S. Cohn, K. M. Hazelwood, V. Vladimirov, and M. Bach. Dynamic program analysis of Microsoft Windows applications. In *ISPASS*, pages 2–12, 2010.

[89] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, Apr 2009.

[90] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 25–25, Berkeley, CA, USA, 2001. USENIX Association.

[91] G. E. Suh, J. Lee, , and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of ASPLOS-XI*, Oct. 2004.

[92] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[93] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. *SIGPLAN Not.*, 44(3):109–120, 2009.

[94] D. Uluski, M. Moffie, and D. Kaeli. Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33:90–98, March 2005.

[95] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[96] G. Vasiliadis and S. Ioannidis. Gravity: a massively parallel antivirus engine. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 79–96, Berlin, Heidelberg, 2010. Springer-Verlag.

[97] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *14th International Conference on High-Performance Computer Architecture (HPCA-14 2008), 16-20 February 2008, Salt Lake City, UT*, 2008.

[98] Wikipedia: Heap Spraying. `http://en.wikipedia.org/wiki/Heap_spraying`.

[99] G. Wondracek, P. M. Comparetti, C. Krügel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California*, 2008.

[100] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, pages 2–7, 1991.

[101] Y. Yang, S. Zhu, and G. Cao. Improving sensor network immunity under worm attacks: a software diversity approach. In *MobiHoc '08: Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, pages 149–158, New York, NY, USA, 2008. ACM.

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

[103] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA, 2007. ACM.

[104] A. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *Networked Systems Design and Implementation (NSDI)*, 2007.

[105] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[106] Security Focus Vulnerability Notes, (http://www.securityfocus.com), Bugtraq ID NNN. http://www.securityfocus.com/bid/NNN/discussion/.

[107] ATmega128 8-bit AVR microcontroller, 2008.