On Information Flow for Intrusion Detection: What if Accurate Full-system Dynamic Information Flow Tracking Was Possible?

Mohammed I. Al-Saleh
University of New Mexico
Department of Computer Science
Mail stop: MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131
alsaleh@cs.unm.edu

Jedidiah R. Crandall
University of New Mexico
Department of Computer Science
Mail stop: MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131
crandall@cs.unm.edu

ABSTRACT

Current intrusion detection systems (IDSes) fall into two very limiting categories: appearance-based or behavior-based. These rely on specifying good *vs.* bad behavior in terms of patterns in the malicious input or in the trace of execution during the attack. Some successful IDS systems have specified attacks in terms of information flow and the influences data sources have on the system, but only in very limited domains such as control data attacks, and typically using information flow tracking mechanisms customized to their purpose. Intrusion detection based on a general method for information flow tracking would allow for very explicit and general definitions of attacks that precluded entire categories of vulnerabilities and exploits, but our current methods for dynamic information flow tracking (DIFT) are inadequate to make this a reality.

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, except in an ad-hoc, application-specific fashion. As a first step toward making information flow a new paradigm for intrusion detection, 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. This coarse definition is not precise enough to detect attacks but serves as a demonstration of our approach to DIFT. 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 intuitive, meaningful results.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection–*Information flow controls*

NSPW'10, September 21–23, 2010, Concord, Massachusetts, USA.
Copyright 2010 ACM 978-1-4503-0415-3/10/09 ...$10.00.

General Terms

Security

Keywords

dynamic information flow tracking, quantitative information flow, intrusion detection

1. INTRODUCTION

In this paper, we present a new dynamic information flow tracking (DIFT) technique, but do so within the context of a 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?** While our DIFT system is far from ideal, it has enough support for challenging types of information flow to enable us to define attacks in a very general way and then demonstrate that our information flow measurements are meaningful and intuitive. Our specific contributions are the following:

- We explore the idea of intrusion detection based on information flow using a prototype DIFT system that can track address and control dependencies.
- We identify two additional specific challenges that DIFT systems of the future will need to address: write-once-read-many-times (WORM) memory locations and looping constructs.
- We provide measurement results characterizing how much information flows from the network to the control path of the CPU for different types of network attacks.

Information flow is one of the most well-studied topics in computer security and privacy, and the idea of applying the ideas of information flow for intrusion detection applications is certainly not new. Why revisit these issues? Our argument for reconsidering the role of information flow in intrusion detection centers around the following two assertions:

1. If information flow tracking is made more accurate and practical, there is the potential for a paradigm shift in the way we think about and implement intrusion detection systems.
2. In the context of integrity as it is applied to intrusion detection, as opposed to confidentiality, information flow tracking can be much more accurate and practical than previously thought.

The first assertion is based upon the second, because as information flow tracking becomes more accurate and practical, intrusion detection systems will use it. We discuss this more in Section 9. Imagine that we had an oracle that gave completely accurate results about any information flow in any system we might be interested in, at zero cost, so long as the threat model did not allow the attacker to execute arbitrary code with the express purpose of removing the tags. Would be continue to define intrusions using appearance-based or behavior-based signatures and profiles, or would we define them based on the changes in information flow that they cause so that we could take advantage of the oracle?

The second assertion is what most of this paper, including the results we present in Section 5, is about. While the prototype DIFT system we have developed, in its current implementation, is neither accurate nor practical enough to cause a paradigm shift, the differences between the DIFT technique we propose in this paper and previous approaches illuminate many challenges. Some of these challenges stretch our current understanding of how information flows in real systems but all of the challenges are tractable and can be addressed in future research.

In particular, we propose a relaxed static stability approach to DIFT where data is tagged with a fixed point number indicating a degree of taint, specifically the mutual information between the tagged data item and some tainted source. More details are in Sections 2 and 3. The relaxed static stability approach allows for "best-effort" tracking of information flow that relaxes the typical conservative notion that information either flows or it does not. What our results in Section 5 show is that all of the potential inaccuracies in propagating taint come from two sources: equivocation caused by operations and joint entropy between the inputs to operations. The latter is particularly interesting because it creates dependencies between operations that may span long periods of time. Joint entropy between inputs manifested as two particular sources of instability in our experiments: looping structures and write-once-read-many-times memory locations. What our results demonstrate is that the path forward for accurate DIFT is to understand equivocation caused by operations and joint entropy between the inputs to operations at a fundamental level and then to address these sources of inaccuracy in a general fashion.

The historical influence of confidentiality applications on information flow research has hampered the application of information flow to integrity and availability applications, and in particular intrusion detection systems. On what grounds do we claim this? After all, quantitative information flow has been the subject of study for several decades [53, 27, 5, 4, 6, 35, 13, 20, 40], and applying information flow ideas in integrity settings to detect attacks is not new [47, 15, 17, 40]. Furthermore, from a purely mathematical point of view integrity is the dual of confidentiality so the results from one should carry over to the other, and intrusion detection can entail confidentiality concerns, in addition to integrity or availability. However, the differences between confidentiality and integrity go well beyond the mathematical, as illustrated by the differences between Biba's model for integrity [3] and those of Lipner [33] or Clark and Wilson [9]. The latter models include notions such as separation of duty, separation of function, and auditing specifically because integrity applications demand a more practical definition of the trust relationships in the model than do traditional confidentiality applications, such as multi-level secure (MLS) applications. In a typical MLS system, full trust is placed in the labels that data has, and no trust is placed in the source code that the system is running. This leads to the necessity for a conservative notion of information flow where either information flows or it does not. For MLS systems a graduated notion of how much information flows is useless given these trust relationships—unless the information flow tracking is 100 percent accurate for all possible programs. For many practical integrity scenarios such as intrusion detection, however, some amount of trust can be placed in the source code being executed, and this means that the labels for data items can be approximations. How to make these approximations as accurate as possible while still remaining practical has been the subject of much less research than is warranted, we believe.

In this paper, 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 [40], we make no claims in this paper 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 *with our current DIFT prototype*. Our aim is to shed light on new directions for DIFT research to support a new paradigm in intrusion detection.

Our claims are the following:

- The relaxed static stability approach to DIFT that we present in this paper offers a general and promising way to deal with implicit information flows such as control dependencies and address dependencies.

- Our results demonstrate that our current implementation gives intuitive results for full-system information flow tracking that match intuitions about how much information actually flows.

- Our results demonstrate that address and control dependencies are critical to track for full-system DIFT.

- Our results illuminate several specific challenges for future DIFT research and suggest a path forward.

We do not claim any of the following:

- We do not claim that our current implementation for tracking information flow from the network into the control path of the CPU is a viable intrusion detection technique without further research. In particular we do not establish in this paper that there are observable differences in the magnitude of information flow between normal traffic and attacks in this particular context.

- We do not claim that our current implementation of relaxed static stability DIFT is accurate enough to cause a paradigm shift in intrusion detection, only that further research into accurate and practical DIFT has this potential.

This paper is organized as follows. First, Section 2 discusses the limitations of current DIFT systems. Then, we describe our relaxed static stability DIFT system implementation in Section 3. This is followed by Section 4 that explains our evaluation methodology, and then our results in Section 5. A discussion of performance issues and applications of DIFT to intrusion detection is in Section 6, followed by a description of the discussion at the workshop in Section 7. Then we discuss related work, followed by the conclusion.

2. ON THE LIMITATIONS OF DIFT

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 paper 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 [47, 15, 41, 12, 44] 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 [47, 15, 41, 12, 44].

Systems that measure information flow and do track address and/or control dependencies are either unstable [22, 23, 50, 49] or have limited, application-specific support for these dependencies. Panorama [56], for example, demonstrates the power of full-system information flow measurement, but handles address and control dependencies in an application-specific manner (see Section 8 for details).

Suh *et al.* [47] 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 paper is a DIFT system that does track address dependencies in a general way.

The DIFT system we present in this paper 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 [47, 46] 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 [15, 16, 44]. 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 [47, 46], and is not solved by information flow tracking systems designed for confidentiality, such as Fenton's Data Mark Machine [22, 23], RIFLE [50], or GLIFT [49], because they also "overtaint" without static analysis, which is impractical for most DIFT applications. **As mentioned previously, the historical influence of confidentiality applications on information flow research has hampered the application of information flow to integrity and availability applications, and in particular intrusion detection systems.** Many practical applications of DIFT have very different requirements than those in which the foundation of early information flow research was laid.

In this paper, we take a first step toward addressing 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 cannot execute arbitrary code with the express purpose of tampering with the DIFT system, either because they or do not know about the DIFT tracking or do not yet have that level of control over the system. 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 require 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.

3. 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. Specifically, this number represents the mutual information between the data object the taint tag is associated with and the taint source, which could be the network, CD-ROM, or some other source. 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 information in an information theoretic sense from 0.0 to 32.0). We chose this range because in our DIFT system the taintedness of a 16-bit or 32-bit word in memory is stored and loaded from its least significant byte for 16-bit and 32-bit operations. 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 6.

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. Note that, while 16-bit integers carry the same amount of information as 16-bit fixed point numbers, the multiplication operation—which is critical in information-theoretic interpretations for our tags—is defined differently for fixed point *vs.* integer representations of numbers. Rather than redefine the multiplication operation for integers, we chose to interpret the tags as fixed point so that the existing defined multiplication operation matches the information-theoretic definition for the tags.

For our results in this paper, 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 (what instructions are executed, their operands, the data that conditional control flow decisions are based on, and the addresses instructions are fetched from). By measuring the information flow over time between this source and sink we can estimate how much control an attacker has over what the CPU is doing, even if their inputs are high-level shell commands or script code. In this paper we are only interested in if these measurements match our intuitions, and make no attempt to detect attacks based on the measurements.

To implement the source and sink, we taint all data that comes from the network card of the virtual machine with 8.0 bits per byte. Using a moving average filter, we plot the amount of tainted data that is used in the control path of the CPU. Our implementation is built on top of Argos [44], which is itself built on top of the QEMU emulator for the x86 architecture. Our implementation handles the five dependencies defined by Suh *et al.* [47], but before describing how each dependency type is handled we must define measurement accuracy in terms of information theory, and also there are two specific research challenges that we identified that place special requirements on how taint propagation is handled.

3.1 Measurement accuracy

The fixed point tags in our DIFT system represent mutual information between the data object the tag is associated with and the taint source, not entropy. This distinction is important for defining measurement accuracy.

Since QEMU turns the x86 instruction set into a load/store architecture we can consider computation dependencies and address dependencies separately. Here we will define upper and lower bounds for taint propagation for the five dependency types. $I(x;y)$ denotes the mutual information between variables x and y, and here y is the destination so that $I(y;\tau)$ is the destination tag we need to calculate for taint propagation.

Copy dependency: If the value of a data object x is copied to a data object y, *e.g.*, in a register-to-register, register-to-memory, or memory-to-register operation, then by definition $x = y$. Thus, simply copying the tag is accurate from an information-theoretic perspective. Note that this would not be true if the tag were interpreted as entropy rather than mutual information with the taint source. Define τ to be the taint source. For copy dependencies, the equality that must hold is:

$$I(x;\tau) = I(y;\tau)$$

Computation dependency: If a data object y is computed as a function of one or more other data objects, *e.g.*, w and x, its mutual information with the taint source depends not only on the mutual information of the inputs with the taint source but also on the mutual information between these inputs ($I(w;x)$). That is, if two inputs are based on different source data we should add their taint values, but if there is the minimum joint entropy between them we should take the maximum value ($\max\{I(w;\tau), I(x;\tau)\}$) instead of addition. Furthermore, some operations destroy information, such as $y := w/x$ or $y := x - x$, which we can model as an additional factor α_1 where $0 \leq \alpha_1 \leq 1$, since this leads to a formula that captures the full range of conditional mutual information. Without loss of generality for only one operand or more than two operands, for a computation (such as an add, multiply, *etc.*) with two source operands w and x, *i.e.*, $y = f(w, x)$, we should maintain the inequality:

$$\alpha_1(\max\{I(w;\tau), I(x;\tau)\}) \leq I(y;\tau) \leq \alpha_1(I(w;\tau) + I(x;\tau))$$

Note that the conditional mutual information $I(X;Y|Z)$ is bounded by $0 \leq I(X;Y|Z) \leq I(X;Y)$, so that any operation that destroys information can be captured with the value $0 \leq \alpha_1 \leq 1$. While α_1 models the equivocation, β_1 can be defined to model the range of possible mutual information values for the inputs that the above inequality represents, where:

$$I(y;\tau) = \alpha_1(\max\{I(w;\tau), I(x;\tau)\}) + \beta_1((\alpha_1(I(w;\tau) \\ +I(x;\tau)) - \alpha_1(\max\{I(w;\tau), I(x;\tau)\}))$$

Thus, the range $0 \leq \beta_1 \leq 1$ captures the full range of the inequality for computation dependencies.

Load and store address dependencies: If the value of a data object x is copied to a data object y and either of them is a memory location that was accessed using address a, then the joint entropy between a and x as well as the entropies $H(a)$ and $H(x)$ are relevant to the taint propagation calculation. Unfortunately, it is not practical to attempt to measure $H(a)$ or $H(x)$. For example, to measure $H(a)$ would require an aliasing analysis to know what addresses a could have pointed to, and a calculation of the joint entropies between all values that could have been loaded and the taint source. In other words, equivocations for address dependencies occur when there is not perfect entropy in the range of values that could have been loaded or stored, due to limitations on the address or redundancy in the values in memory. We can define the inequality:

$$\alpha_2(\max\{I(a;\tau), I(x;\tau)\}) \leq I(y;\tau) \leq \alpha_2(I(a;\tau) + I(x;\tau))$$

Again, $0 \leq \alpha_2 \leq 1$ models the equivocation, or destruction of information, and β_2 models the range of mutual information between the address a and the value x, with $0 \leq \beta_2 \leq 1$:

$$I(y;\tau) = \alpha_2(\max\{I(a;\tau), I(x;\tau)\}) + \beta_2((\alpha_2(I(a;\tau) \\ +I(x;\tau)) - \alpha_2(\max\{I(a;\tau), I(x;\tau)\}))$$

Control dependencies: If a data object y is set to a particular value based on a conditional control flow transfer (such as an `if` statement or a `for` loop) that was conditioned on the value of a data object x, then the amount of information that flows from x to y depends on all of the possible executions of the program that *could have happened*, but will always be bounded by the inequality:

$$0 \leq I(y;\tau) \leq \alpha_3(I(x;\tau))$$

Again, $0 \leq \alpha_3 \leq 1$ models the equivocation, which occurs when possible program executions give the same output for a particular variable despite the control dependency. We define β_3, where $0 \leq \beta_3 \leq 1$, such that:

$$I(y;\tau) = \beta_3((\alpha_3(I(x;\tau)))$$

β_3 represents mutual information between x and the program counter, particularly when there had already been a recent condition check on x.

For the open-loop controller, we defined $\alpha_1 = 1$, $\alpha_2 = 1$, $\beta_1 = 0$, and $\beta_2 = 0$. The quantity $\beta_3\alpha_3$ was modeled as a pair of decreasing geometric series' to ensure that taint value assignments that happened sooner after the conditional check were tainted more, as well as to address instability problems that were due to loops. In general, α values are associated with equivocation, or destruction of information, and β values are associated with mutual information between the inputs. Thus, for computation and address dependencies our implementation basically assumes no equivocation but the minimum amount of joint entropy (or maximum amount of mutual information) between inputs. This means that the taintedness is spread out over a larger amount of data. Basically, by making conservative estimates of how much information is copied and ignoring equivocation of individual operations, we are able to taint as much data in the system as possible while maintaining stability by not tainting it as much. Through empirical measurement, we found this approach to be the best tradeoff between accuracy and stability for the purpose of this paper, which is to show information flow measurements that match our intuitions about how information flows for different attacks.

3.2 Specific challenges

During the early development and testing of our system, we identified two sources of instability: memory locations that are written once and read many times, and looping constructs.

3.2.1 WORM memory locations

In our experiments many values are stored to memory once and then read many times, becoming endless sources of taintedness. We refer to this here as Write-once-read-many-times (WORM) memory locations. We found that forcing the taintedness of values that sit in memory to decay over time is essential for DIFT system stability, thus every time a memory location is read its taint value is multiplied by a constant factor of 0.99 and stored to both the source and the destination. 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.

3.2.2 Looping constructs

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 1:

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

As described later in this section, for 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 when the value of y is set, with an appropriate amount of throttling of the amount of taintedness. This has the desired effect from an information flow perspective, and is similar in concept to Fenton's Data Mark Machine [22, 23]. 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 taintedness, then without throttling condition check taint values each element of the array Y will be tainted with 3.5 bits. This means that from 3.5 bits of taintedness we will have generated 3500 bits of taintedness. This is not necessarily incorrect from an information-theoretic perspective because the taint tags represent mutual information, not entropy, but we found this to be a source of instability. 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 $\frac{1}{1-0.99} \times I(x; \text{Taint Source}) = 100 \times I(x; \text{Taint Source})$ (in this case, 350) bits of taint is generated in Y by the conditions on x. Thus looping constructs are 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 [34] will be of critical importance.

3.2.3 Interpretation of instabilities

The write-once-read-many-times memory locations are an artifact of our choices for α_1, α_2, β_1, and β_2. These choices are only stable when data is not copied many times before being destroyed. For write-once-read-many-times memory locations an additional throttle is required to maintain stability.

The loop construct instability is associated with β_3 in that, implicitly, for a loop the conditions on the loop guards can be viewed as making copies of the result through repeated, related comparisons. This is why we address loop construct instability by throttling taintedness for checks on condition flags.

Both sources of instability are due to the fact that information is often copied many times in a system before being destroyed.

3.3 Taint propagation

Now we describe how taint tags are propagated for the different types of dependencies. Where multiple dependencies are involved, we use the maximum operation *max* to determine the destination taint value.

3.3.1 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*, for reasons explained above having to do with write-once-read-many-times memory locations. Note that address and control dependencies can also affect copy operations.

3.3.2 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 control dependencies can also affect computation operations, but address dependencies cannot since QEMU instruction emulation uses a load/store instruction set.

3.3.3 Load and store address dependencies

If the address of a load or store operation 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.

3.3.4 Control dependencies

When tainted data is used for conditional control flow decisions (*e.g.*, `w` and `x` in the C code "`if (w == x)`" 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, *i.e.*, *max*$\{w, x\}$. 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 a constant factor of 0.5, 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. If a tainted flag is used and its taint value is copied to the program counter, the taint value of the flag is reduced by multiplying it by the constant throttle factor of 0.99 for reasons explained above having to do with instability due to looping constructs. 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 m of the program counter's taintedness e, using the equation $m' = cm + (1 - c)e$ with $c = 0.999511719 = \frac{2047}{2048}$. (This is the y-axis for all figures in Section 5.)

3.4 Additional considerations

The above design decisions are the result of extensive testing to develop a controller that is both stable and sensitive to the information flow being measured. 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 address 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.

The 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.

4. 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.

Vulnerability	bugtraq ID [59]
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 1: Attacks we tested our DIFT system with.

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 5, 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 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 [47, 15, 41, 12, 44]. 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 [52]. We purposely chose a diverse variety of attacks for testing to demonstrate the power of relaxed static stability DIFT to measure 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.

5. 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.

5.1 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 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. 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 [47, 15, 41, 12, 44] 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 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

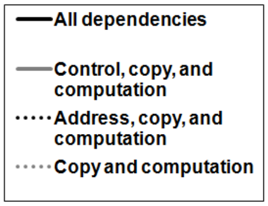


Figure 1: Legend for vulnerability results.

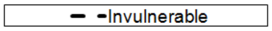


Figure 2: Legend for invulnerable baseline results.

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.

5.2 Basic results

Figure 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 4.

Figure 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.

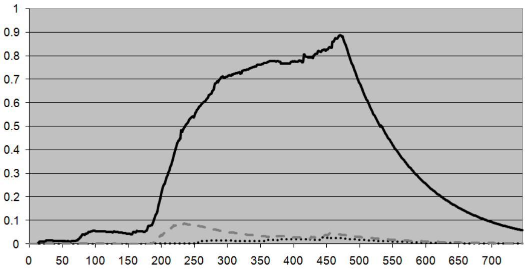


Figure 3: Code Red attacking a vulnerable version of Windows 2000 and IIS web server.

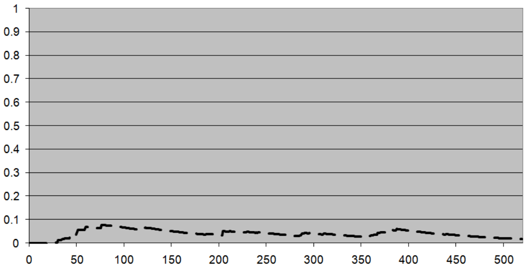


Figure 4: Code Red attacking an invulnerable version of Windows 2000 and IIS web server.

5.3 Repeatability

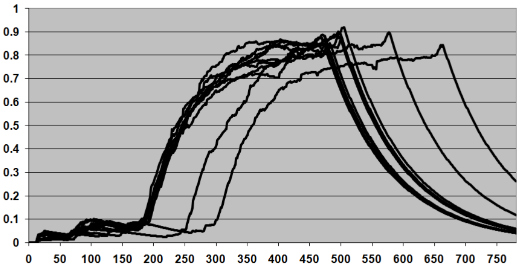


Figure 5: Code Red attacking a vulnerable VM, repeated.

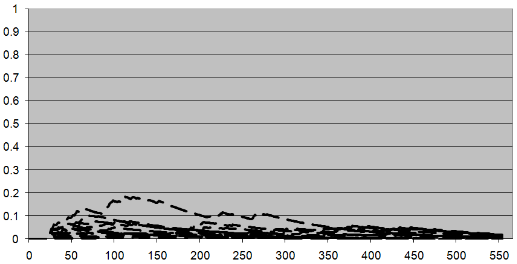


Figure 6: Code Red attacking an invulnerable VM, repeated.

Figures 5 and 6 show the same tests as Figures 3 and 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.

5.4 Importance of implicit dependencies

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

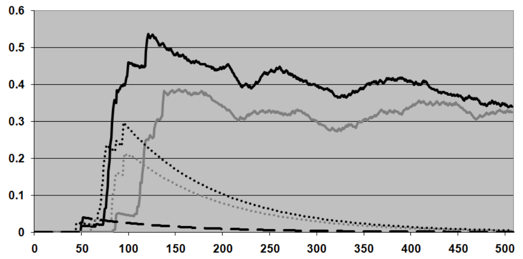


Figure 7: 9633 (ASN.1 Integer Overflow).

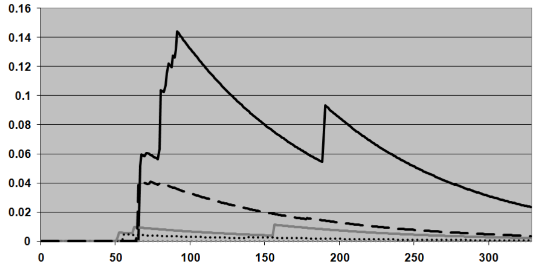


Figure 9: 8205 (DCOM RPC Buffer Overflow).

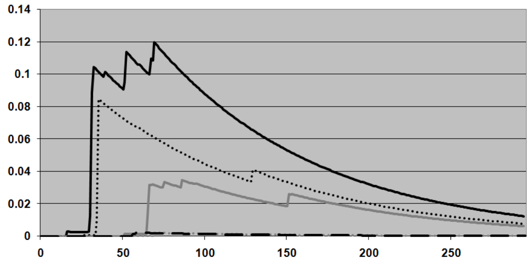


Figure 8: 13300 (ASN.1 Heap Corruption).

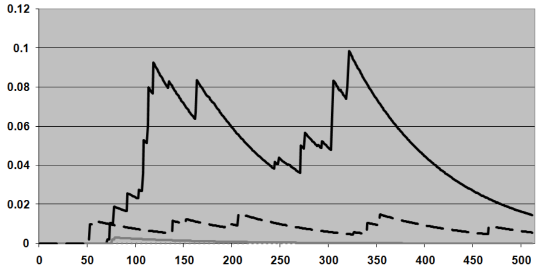


Figure 10: 12096 (Nullsoft SHOUTcast Format String).

to demonstrate the concept of relaxed static 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 paper 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.

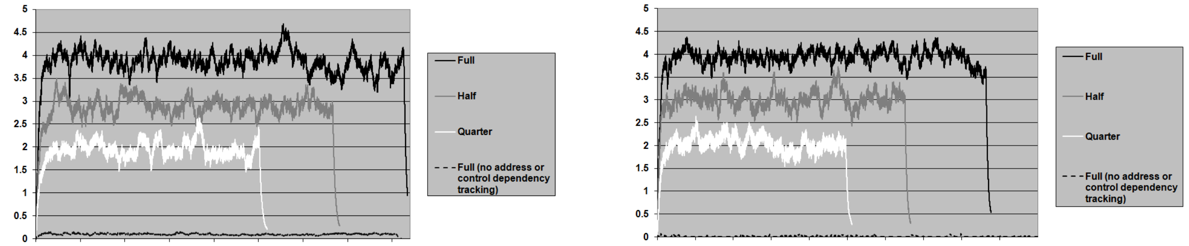
Figures 7 and 8 show the results for exploits for two different vulnerabilities in the ASN.1 parser in Windows. It is interesting that Figure 7 shows control dependencies as a major driver of the information flow while Figure 8 shows the same for address dependencies. ASN.1 is an XML-like format for 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 [47, 15, 41, 12, 44] will not detect when control flow is hijacked because the exploit overwrites a pointer to a function pointer, not the function pointer itself, meaning that there is a 32-bit address dependency in the corruption of the control data.

Figure 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 10 shows the same, and Figures 11 and 14 also show results where the vulnerable and invulnerable tests are indistinguishable unless both address and control dependencies are tracked. Figure 12 is interesting in that the information flow is driven almost entirely by address dependencies.

There are two interesting aspects to Figure 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 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 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.

5.5 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,



(a) Normal

(b) Compressed

Figure 15: Measured information flow for controlled rates.

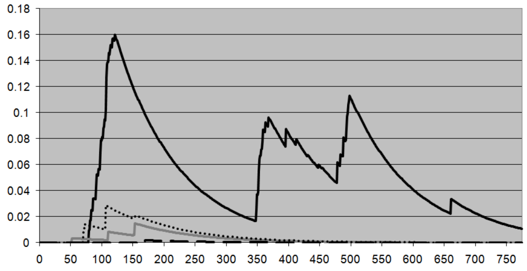


Figure 11: 5310 (MS-SQL Resolution Service Heap Overflow).

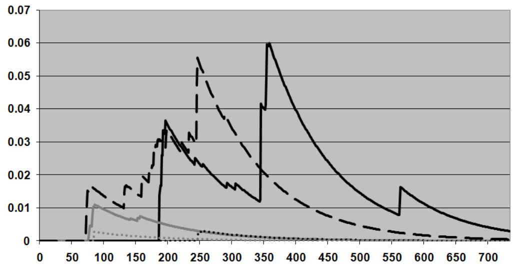


Figure 13: 10108 (LSASS Buffer Overflow).

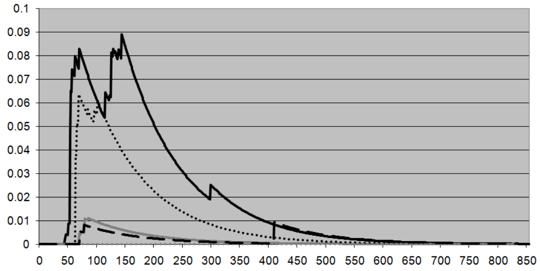


Figure 12: 5411 (MS-SQL Authentication Buffer Overflow).

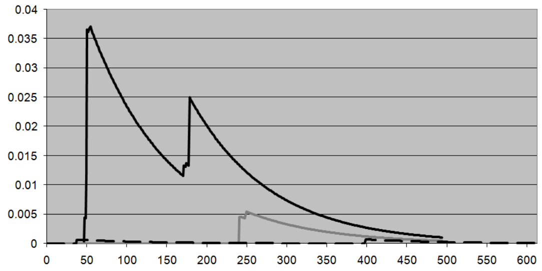


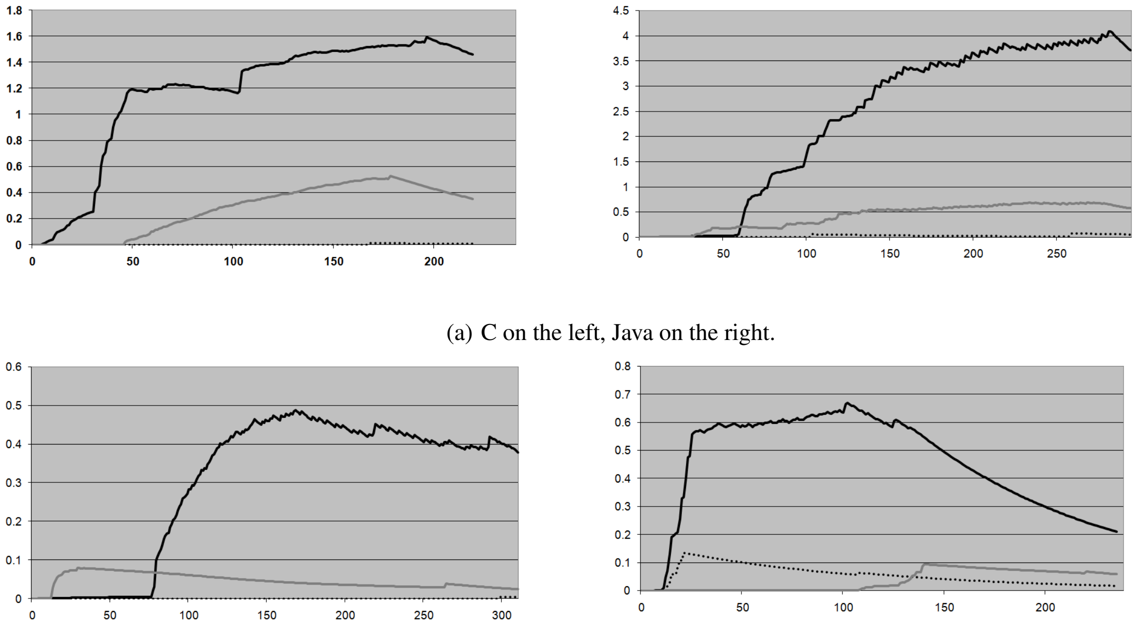
Figure 14: 11372 (NetDDE Buffer Overflow).

we fixed the rate at which we did this at 0.8 repetitions 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 15(a). Figure 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 15(a), we simply copied each raw binary and executed it. For the second set of experiments, shown in Figure 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 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 15(a) and 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.

5.6 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.



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

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

Figure 16: High-level languages.

Actual Rate (Controlled)	Uncompressed Measurement (Figure 15(a))	Compressed Measurement (Figure 15(b))
Full (1.0)	1.0	1.0
Half (0.50)	0.60	0.58
Quarter (0.25)	0.31	0.30

Table 2: Rate measurement integrals over time, normalized to full rate.

Figures 16(a) and 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.

5.6.1 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 17(a) and 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 [52] 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.

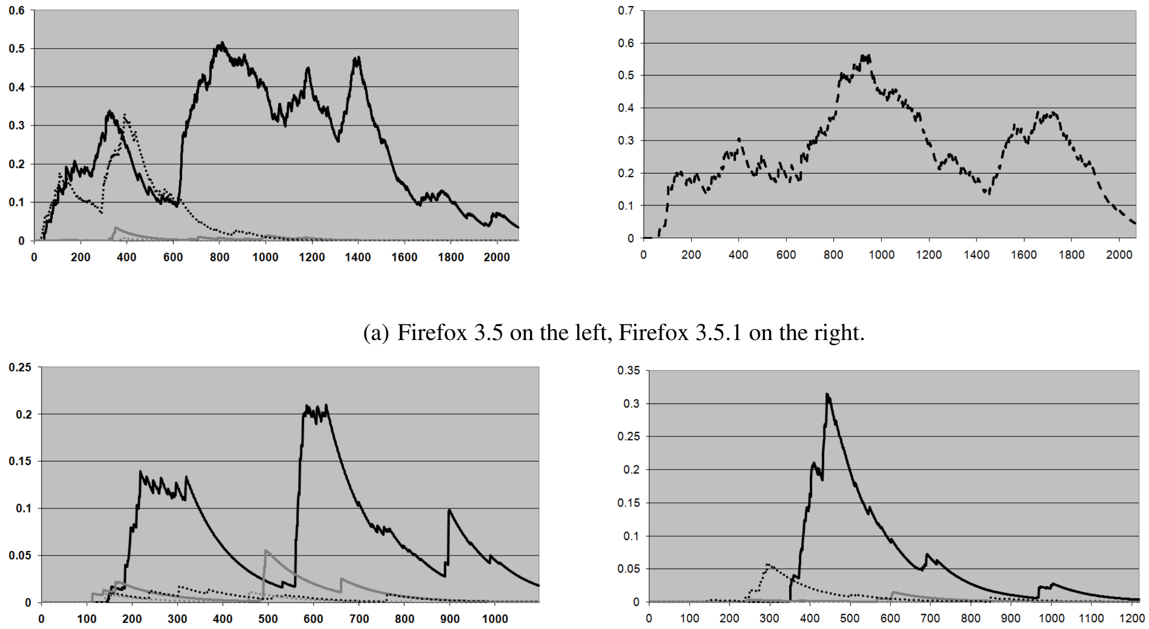
6. DISCUSSION AND FUTURE WORK

Here we discuss some issues related to the performance *vs.* accuracy tradeoff and applications of DIFT to intrusion detection.

6.1 Performance *vs.* accuracy

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 [25], 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 [47, 15], and designed for flexible policies [19, 45] or decoupled on a separate coprocessor [28]. Many of these systems perform some form of compression to reduce storage overhead and memory bandwidth usage. On-demand emulation [26] 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 [42]. 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



(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 17: Heap-spraying browser exploits.

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.

6.2 Intrusion detection applications

Most intrusion detection systems fall into one of two paradigms: appearance-based or behavior-based. These terms come from Cohen [11], who also observed that "information only has meaning in that it is subject to interpretation." Attacks typically take the form of malicious inputs that are interpreted in some way by the system that leads to the attacker gaining increased control in order to violate security. Appearance-based detectors look for byte patterns that will be interpreted in a way that leads to malicious operations, behavior-based detectors look for artifacts of malicious behavior in the trace of operations carried out by the system. Neither of these paradigms has been able to handle obfuscation techniques in a way that precludes new attacks. See, *e.g.*, Szor [48] for a discussion of the co-evolution of virus obfuscation and anti-virus techniques.

If information flow tracking were made more accurate and practical through further research, then defining intrusions in terms of the flows of information that are considered malicious would offer a more general definition that precluded whole classes of attacks at a time. Such mechanisms have been successful even at a very coarse definition of information flow, for example Tripwire [29]. The early applications of DIFT [47, 15, 41, 12] were very effective at catching remote memory corruption attacks, but offer no guarantees in a stronger threat model where an attacker seeks to evade detection due to address and control dependencies. In general, from phreaking to memory corruption to web application attacks, exploits always take that form of management information being stored "in-band" by the system and the attacker causing their inputs to be interpreted as management information [1]. This fact has been exploited in past intrusion detection systems using information flow tracking mechanisms designed for their specific purposes, but **what is needed for a paradigm shift is a general information flow tracking mechanism that supports defenses against any general class of attacks that can be defined in terms of information flow.**

We see information flow for intrusion detection being applied in two distinct ways: as a honeypot technique when the attacker does not anticipate DIFT-based intrusion detection, and as a general technique that detects attacks before the attacker has control over what code a user's machine is running. The definition of attack we present in this paper falls into the former category (of course, more research is needed to do the profiling and address false positives and false negatives). Examples of applications that fall in the latter category might include DIFT systems that enforce separation in browsers or that ensure that PDF inputs affect only the rendering of the displayed document and not any other aspects of the PDF rendering process. A key research question will be how much trust to place in the code being executed, since the code being executed may be interpreting high-level Javascript or PostScipt from the attacker. **Will the modern trend toward live content lead information flow for intrusion detection applications into the same traps as confidentiality and MLS systems, or are the trust relationships still fundamentally different in a way that researchers can exploit to our advantage? We leave this as an open question for the discussion at NSPW.**

7. DISCUSSION AT THE WORKSHOP

Discussion at the workshop had three major themes: (1) previous work that, while not directly related to dynamic information flow tracking, could provide some valuable insights in terms of quantitative approaches to information flow; (2) how the performance and threat model aspects of our proposed approach may apply more readily to data provenance than to intrusion detection; and, (3) ways that our definition of taint could be modified to better model information flow.

While there has not been a lot of work on quantitative, dynamic information flow tracking, one NSPW attendee suggested that some of the efforts to quantify covert channels [53, 24, 31, 37] may be instructive in this regard. It was pointed out that the basic tradeoff of conservative approximation is similar to our approach. We plan to investigate the connection between quantifying covert channels and DIFT further. Also, another attendee pointed out generalized noninterference [36] as another area of research where similar tradeoffs were made. This could also be very instructive for future work on dynamic information flow tracking.

Another topic of discussion was the performance overhead of and the threat models that could be supported by our proposed DIFT system. The general consensus was that areas such as forensics and data provenance may be more promising goals for application of relaxed static stability DIFT in the near future. Intrusion detection often requires a stronger threat model and near-native performance, so DIFT applications in this area may be limited until advances in relaxed static stability DIFT's performance and accuracy are made. We concur, and do plan to focus on forensics and data provenance for the near future.

Another interesting part of the discussion at the workshop was our definition of taint and what information is kept in the taint mark. Currently, our taint marks are fixed-point numbers that represent the mutual information between the data that is tagged and some tainted source. As one attendee pointed out, this one-dimensional taint mark makes it impossible to know the mutual information between two different taint marks, which is one of our major sources of over-tainting. Having a vector of some kind or some additional information in the taint mark is definitely something we plan to explore. It was also suggested that taint marks could be interpreted as fuzzy logic instead of mutual information. This suggestion is particularly interesting in light of something that one of the reviewers of our NSPW submission pointed out, which is that the interpretation of taint marks as mutual information gives a false sense of accuracy which may hinder efforts for approximating the taint mark. How important is it to have an information-theoretically correct way to combine two taint marks when those taint marks themselves are already approximations?

We are very grateful for these helpful comments and plan to explore all of these issues that were raised in future work.

8. RELATED WORK

DIFT in the context of detecting attacks was introduced by Suh *et al.* [47]. TaintBochs [8] was another early application of tainting that was applied to analyzing data lifetime in a full system. The Minos project [15, 17] explored various policy tradeoffs for DIFT schemes and higher-level systems issues such as virtual memory swapping, TaintCheck [41] explored some of the issues of using DIFT for end-to-end detection of exploits for vulnerabilities in commodity software, and Vigilante [12] employed DIFT for automated worm defense. Argos [44] 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 [56, 55, 2], network protocol analysis [32, 54], and dataflow tomography [39]. 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 [56] 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 [17, 46], especially when combined with store address dependencies.

The earliest DIFT papers [47, 15] identified the problems with address and control dependencies. More details and analysis of these issues followed [14, 7, 18, 43], including a recent quantitative analysis of full-system pointer tainting [46]. Flexible tainting schemes that allow taint policies to be specified [47, 41, 51, 45, 19, 10] 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 [22, 23, 50, 49] 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 noninterference, 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 [22, 23] 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 [50] 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 [49] 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) [38] is a concept based on information flow that relaxes the requirement of noninterference while still making confidentiality and integrity guarantees. Several systems [21, 58, 30] 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.* [40] 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 [35] 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 [57] 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.

9. 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 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. At NSPW we intend to 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?**

Why do we believe that if information flow tracking is made more accurate and practical, there is the potential for a paradigm shift in the way we think about and implement intrusion detection systems? Our argument is the following. Appearance-based and behavior-based intrusion detection techniques look for patterns in data that is passed around a system or in the sequences of operations that occur at the system interfaces, such as APIs and system call interfaces. This means that what is malicious behavior *vs.* what is normal behavior has to be defined for one place in the system: either a specific place where data will be stored or a specific interface. Since malicious behavior is a global system property and information only has meaning in that it is subject to interpretation, this means that traditional intrusion detection signatures must capture a global property (how that data or sequence of operations will be interpreted semantically) using limited local information. Practical, full-system dynamic information flow tracking will allow IDS designers to define malicious *vs.* normal behavior using global information, so that the definition can be both precise and general.

As an illustrating example, consider a complex API that allows a process to make modifications to the system configuration in a practically innumerable variety of ways. Contrast the following three approaches to defining what constitutes malicious modifications to system configuration:

1. **Appearance-based**: Inside a userspace process, the patterns of executable machine code are defined to be malicious or not as a property of the string.

2. **Behavior-based**: At the system call API, the patterns of system calls are defined to be malicious or not as a property of the sequence of operations.

3. **Information-flow-based**: Within the operating system kernel and filesystem, where the system configuration is actually stored, flows of information are defined to be malicious or not as a property of the information that flows between objects.

We argue that the third approach is more powerful in terms of precisely defining malicious behavior in a general way that precludes entires classes of behavior. Defining behaviors in terms of information flows is fundamentally different from combining patterns of operations from many different interfaces, since patterns from multiple interfaces does not address the problem that the semantically-equivalent ways to achieve something can be combinatorial. Also, the third approach could make it possible that the threat model for information flow tracking is realistic, since dynamic information flow tracking could happen in the kernel, or perhaps in a hypervisor, so that the relevant code that causes the information flows being tracked is existing system code known not to be malicious. We will explore this idea further at NSPW, and hope that the reader will propose their own ideas for what information-flow-based intrusion detection systems might look like.

Acknowledgments

We would like to thank the NSPW reviewers for their insightful comments. Also, our shepherd, Matt Bishop, provided very valuable feedback which is reflected in the paper, and the NSPW attendees were helpful and encouraging and raised some interesting questions that require further research. Discussions with Rafael Fierro about control theory were also very helpful. This work was supported in part by the U.S. National Science Foundation (CNS-0905177 and CNS-0844880). Crandall's travel to the workshop

[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.

was supported by CNS-1017602. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

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

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

[3] K. J. Biba. Integrity Considerations for Secure Computer Systems. In *MITRE Technical Report TR-3153*, Apr 1977.

[4] R. Browne. The turing test and non-information flow. In *IEEE Symposium on Security and Privacy*, pages 373–388, 1991.

[5] R. Browne. An entropy conservation law for testing the completeness of covert channel analysis. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 270–281, New York, NY, USA, 1994. ACM Press.

[6] R. Browne. Mode security: An infrastructure for covert channel suppression. In *IEEE Symposium on Security and Privacy*, pages 39–55, 1999.

[7] 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.

[8] 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.

[9] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–193, 1987.

[10] 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.

[11] F. Cohen. Computer viruses: Theory and experiments. In *7th DoD/NBS Computer Security Conference Proceedings*, pages 240–263, September 1984.

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

[13] J. R. Crandall, J. Brevik, S. Ye, G. Wasswerann, D. A. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Putting trojans on the horns of a dilemma: Redundancy for information theft detection. Special Issue on Security in Computing of the Transactions on Computational Sciences Journal, Springer Lecture Notes in Computer Science, 2009.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[21] 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.

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

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

[24] J. W. Gray III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.

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

[26] 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.

[27] J. W. G. III. Toward a mathematical foundation for information. *Journal of Computer Security*, 1(3-4):255–294, 1992.

[28] 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.

[29] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM.

[30] 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.

[31] Light Pink Book. A guide to understanding covert channel analysis of trusted systems, version 1. NCSC-TG-030, Library No. S-240,572, November 1993. TCSEC Rainbow Series Library.

[32] 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.

[33] S. B. Lipner. Non-discretionary controls for commercial applications. In *IEEE Symposium on Security and Privacy*, pages 2–10, 1982.

[34] 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.

[35] 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.

[36] D. McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, page 177, 1988.

[37] J. McHugh. Covert channel analysis, 1995.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

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

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

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

[49] 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.

[50] 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.

[51] 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.

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

[53] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *IEEE Symposium on Security and Privacy*, pages 144–161, 1990.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

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