Influence: A Quantitative Approach for Data Integrity

James Newsome and Dawn Song

February 19, 2008
CMU-CyLab-08-005

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Influence: A Quantitative Approach for Data Integrity

James Newsome
Carnegie Mellon University
jnewsome@ece.cmu.edu

Dawn Song
University of California Berkeley
dawnsong@cs.berkeley.edu

Abstract

A number of systems employ dynamic taint analysis *to detect overwrite attacks in commodity software. These systems are based on the premise that low-integrity inputs should not control values such as function pointers and return addresses. Unfortunately, there are several programming constructs that can cause false positives and false negatives in these systems, which are currently handled by manual annotation, ad-hoc rules, or not at all.*

In this work we propose to use channel capacity*, a quantitative measure of information flow, as a quantitative measure of control. When measuring control, we refer to this measure as* influence*. We use influence as a theoretical tool to formally investigate programming constructs known to be problematic for dynamic taint analysis.*

While calculating influence in arbitrary programs is undecidable in the general case, we propose and implement practical techniques for automatically bounding and probabilistically estimating influence in x86 programs. We show that this tool is able to automatically find useful influence bounds in code constructs known to be problematic in dynamic taint analysis. We also use it to analyze a dynamic taint analysis alert in samba, showing that it is a false positive, and another alert in SQL Server, showing that it is a true positive.

1 Introduction

Dynamic taint analysis has lately been a popular technique for enforcing Biba low water-mark [4] data integrity policies in commodity software [7, 9, 20, 21, 23]. In this work, we refer to such systems as Dynamic Taint Analysis for Data Integrity, or DTADI, systems. The premise behind DTADI systems is that low-integrity inputs, such as data read from the network, should not exert control over high-integrity operations, such as dereferencing a function pointer. In such systems, low-integrity inputs are marked as tainted. The output of every program operation is marked as tainted if any operand is tainted, and untainted otherwise. The systems then monitors high-integrity operations to detect when they are about to use tainted data. Such systems have been found to be useful for detecting *overwrite* attacks; for example when a buffer overflow allows an attacker to overwrite a function pointer, that function pointer will be marked as tainted, and the attack detected when the function pointer is dereferenced.

A challenge in implementing DTADI systems is that in real-world programs, low integrity inputs *do* legitimately exert *some* control over high-integrity operations. In such cases, the low integrity input is usually used to select from a relatively small set of legitimate values. If a DTADI system were to propagate the taint attribute for *every* operation where a tainted operand could affect the output value, a large number of integrity violations would be detected that are not actually attacks; *i.e.* the system would have many false positives. For example, a function pointer may be set to point to one of a few request-handling functions, depending on a request-type specified by a low-integrity input.

Current DTADI systems address this challenge by only propagating the taint attribute for operations where the input is deemed to have "a lot" of control over the output. Typically, this means *direct* assignment, *i.e.* data movement and arithmetic instructions. *Indirect* assignments, such as via `if-then-else` structures, are ignored because they typically give the input relatively little control over the value of the assignment[1]. Assignment via indexing operations, such as loading data via a tainted pointer, are also sometimes ignored due to the prevalence of using an untrusted input to calculate an offset into a relatively small table. Considering again the case where a low-integrity input is allowed to select a function-pointer, these taint propagation policies can prevent false positives when the function-pointer is selected via an `if-then-else` statement or loaded from a table, while still detecting if the input is able to directly overwrite the function pointer via a buffer overrun.

There are two fundamental problems with the taint propagation policies in current DTADI systems. The first problem is that they are manually specified in an ad-hoc manner,

[1] Indirect assignments are also relatively challenging to track, as they require control-flow analysis

based on an intuitive understanding of which operands exert "a lot" of control over the result. The resulting policies do not work in every situation, and can grow complex as special cases and exceptions are added to improve accuracy. For example, most current DTADI systems recognize some common program operations where the output does not depend on the input, such as when a register is `xor`ed with itself, or bit-masked with zero. To our knowledge, such exceptions have been added to DTADI systems manually, in an on-demand basis when such operations are found to otherwise cause inaccuracy.

The second problem with the taint propagation policies in current DTADI systems is that they typically operate at a very fine-grained level; *e.g.* individual x86 instructions, or even individual micro-code instructions that the x86 instructions are first broken into. At this level, important interactions between instructions and cumulative effects of instructions are lost. For example, consider the program: `if (a == 0) b := 0; if (a == 1) b := 1; ...; if (a == 255) b:= 255`. At the level of analyzing the individual `conditional-jump`, `jump`, and `move` x86 instructions that this program would be compiled into, there is no interaction between `a` and `b`. At the level of examining an individual `if` structure, `a` appears to have little control over the value of `b`: `a` effectively selects one of two values for `b`. Only by looking at the whole sequence of `if` statements can we see that `a` is effectively copied to `b`.

The result of these problems is that current DTADI systems suffer from both false positives and false negatives. False positives occur when data used in a high-integrity operation is derived from low-integrity inputs, after being constrained by arithmetic manipulation or sanity checks to ensure the resulting value is within an expected range. False negatives occur when low-integrity inputs effectively control the value of another variable, but in an indirect way such as via implicit data flows. The previous example using `if-then-else` statements is such an example.

In this work we show that channel capacity, a measure used to formally quantify how much information flows from one object to another within a program, is also a useful tool for formally quantifying how much *control* one object has over another within a program. In the context of measuring *control* we refer to this measure as *influence*. As a theoretical tool, influence provides a formal basis for thinking about and evaluating current DTADI taint propagation policies. In Section 4, we use influence to formally reason about program structures that are difficult to reason about with current DTADI taint propagation policies.

We also demonstrate methods for *automatically* reasoning about influence in binary x86 programs. Unfortunately, calculating the exact influence of one object over another in an arbitrary x86 program is undecidable in the general case, and computationally expensive for large programs. However, we build a tool that can exactly measure influence in small loop-free programs, and on general x86 programs can be used to calculate useful lower bounds, upper bounds, and probabilistic estimates of influence. This tool does not suffer from the problems of DTADI taint propagation policies of requiring ad-hoc manual propagation rules, and it can in theory be evaluated over arbitrarily large program structures instead of operating at the per-instruction level. The current tool is too computationally expensive to calculate the influence of every low-integrity input over every high-integrity operation in a real-world program, and does not aim to replace DTADI systems. However, it is practical for performing a deeper analysis of alerts generated by DTADI systems. We use it to examine two alerts generated by a DTADI tool on real-world x86 programs, confirming an alert in SQL Server as an integrity violation, and showing that an alert in Samba is likely a false positive. In Section 8 we suggest future directions for using our techniques to improve the accuracy and performance of DTADI systems.

The contributions of this work are:

- We propose channel capacity as a formal quantitative measure of control. In this context, we refer to channel capacity as *influence*.
- We use influence to formally investigate program structures that are problematic for systems that use dynamic taint analysis to enforce data integrity (DTADI).
- We build a tool that can automatically reason about influence (and equivalently, channel capacity) in binary x86-programs.
- We use our influence measurement tool to analyze alerts generated by a DTADI system on real-world software. We find that one is a false positive, and verify the other as a true positive.

2 Related Work

2.1 Dynamic taint analysis for data integrity

A number of systems have been proposed to perform dynamic taint analysis to enforce Biba low water-mark data integrity policies on x86 binary programs, for the purpose of detecting overwrite attacks [7, 9, 20, 21, 23]. While these approaches work well for many programs, they propagate the taint attribute at the instruction level, using ad-hoc rules about which operands exert "a lot" of control over the result of each operation. As a result, none employ general solutions for handling sanity checks, implicit flows, or identity functions. These systems do recognize and handle some common-cases of these problems.

Xu *et al.* [26] implement a system to rewrite the C source code of a program to perform dynamic taint analysis. They detect implicit flows that occur in some C-level if-then-else structures.

2.2 Dynamic taint analysis for data confidentiality

Dynamic taint analysis has also been used by several systems to enforce Bell-LaPadula [3] data *confidentiality* policies [6, 14, 17, 27]. The implementations of these systems are similar to taint analysis systems used to enforce data integrity. Instead of marking low-integrity inputs as tainted and checking whether high-integrity operations use tainted data, these systems mark confidential data as tainted, and check whether tainted data is written to untrusted outputs. These systems propagate the taint attribute based on whether an operand leaks "a lot" of information to the result of an operation. In practice, the taint propagation policies in these systems are quite similar to those in DTADI systems, though they are often tuned to propagate the taint attribute more aggressively. The systems proposed by Egele *et al.* [14] and by McCamant *et al.* [17] also employ some static analysis to account for positive implicit flows, but do not handle negative implicit flows without manual annotation.

McCamant *et al.* [17] use taint analysis to quantitatively bound how much information about a secret input may have leaked. They track the taint attribute at the bit level, as well as keeping a global leakage counter. The global leakage counter is used in several ways; in particular it is incremented when a branch depends on a tainted input, thus accurately bounding leakage due to positive implicit flows. They calculate the upper bound of the number of bits leaked as the number of tainted bits sent to the output, plus the global leakage counter. Negative implicit flows are not accounted for, unless annotations are added to increment the global leakage counter. The total information leakage bound is kept reasonable using some manual annotations to pre-emptively mark bits as leaked at certain program points before their taint attribute can be propagated to a larger number of bits.

2.3 Information flow

There is a large body of work on information-flow security. Sabelfeld *et al.* provide a good survey of the field [22].

Most prior work seeks to detect or prevent *any* flow of sensitive data to an insecure output. Vachharajani *et al.* [24] propose and implement a system to dynamically detect unpermitted information flows in binary programs. Venkatakrishnan *et al.* [25] propose a provably correct system to enforce non-interference for a small well-structured language; they are able to track implicit flows using the structure of their proposed language.

Denning first proposed to quantitatively measure information flow [12], defining the amount of information transferred in a flow as the reduction in uncertainty (entropy) of a random variable. Other seminal work in quantitative information flow was done by Millen [18] and by Gray [15].

Clark *et al.* [10, 11] and Malacaria [16] propose frameworks for measuring how much information is leaked by programs written in simple imperative languages.

3 Influence: quantifying control

Our first step is to formally define a useful quantitative measure of control. We begin by defining a measure we call *influence*, based on an intuitive understanding of *control*. We then show that the influence of one variable over another is exactly equal to the maximum information flow, or *channel capacity*, between those two variables.

3.1 Program model

Let $\mathcal{P}$ be a program, or program structure, that takes a set of inputs, and deterministically computes a set of outputs. We partition the inputs into two sets: `LOW`, which is the set of (low integrity) inputs which we wish to track, and `HI`, which is the set of all other inputs that the program takes. Formally, we have: $\mathcal{P}(\texttt{LOW}, \texttt{HI}) = \texttt{OUT}$. While we allow the domains of `LOW` and `HI` to be infinite, we assume that the domain of `OUT` is finite. Our goal is to characterize how much control `LOW` has over `OUT`.

For simplicity, we represent `LOW`, `HI`, and `OUT`, each as a single scalar variable that $\mathcal{P}$ reads from or writes to. Other forms of input and output can be transformed to fit this model; *e.g.* socket reads and writes can be represented as reading some range of bytes from `LOW` and writing to some range of bytes in `OUT`.

3.2 Influence

Intuitively, how much control `LOW` has over `OUT` is related to the number of different values that `LOW` can cause `OUT` to take on. If `OUT` always takes on the same value, regardless of the value of `LOW`, then `LOW` has no control over `OUT`. At the other extreme, if selecting an appropriate value of `LOW` can cause `OUT` to take on any value in the codomain of $\mathcal{P}$, then `LOW` has maximal control over `OUT`.

We propose to measure the control that `LOW` has over `OUT` as *influence*, which we denote $\mathcal{C}$. We define the influence of `LOW` over `OUT`, given a particular assignment of `HI`, as the log of the size of the range of $\mathcal{P}$ for the given assignment of `HI`. For convenience, we take the log base 2, which allows us to measure influence in bits. Formally:

$\mathcal{C}(\mathtt{LOW}{\rightarrow_{\mathcal{P}}}\mathtt{OUT}|\mathtt{HI}=\mathtt{hi}) =$
$\log|\{\mathtt{out}|\exists\mathtt{low}:\mathcal{P}(\mathtt{low},\mathtt{hi})=\mathtt{out}\}|.$

To show that this is intuitively a useful measure for control, we consider a few simple examples. If OUT takes on the same value regardless of LOW, the influence is zero bits. If selecting LOW can cause OUT to take on any value in the codomain of $\mathcal{P}$, then the influence is exactly the number of bits needed to represent OUT. In cases where LOW is able to freely overwrite n bits of OUT, then LOW has n bits of influence over OUT.

Note however that influence is more informative than tracking a binary attribute for each bit of whether LOW can affect that bit, because it takes dependencies between bits into account. For example, suppose that LOW can cause a 32-bit OUT to have a value of all zero-bits or all one-bits, but no other values. Although LOW can affect the value of all 32 bits of OUT, it has much less control over OUT as a whole than if LOW could cause OUT to take on all 2^{32} combinations of ones and zeroes. The influence measure reflects this; since LOW can only cause OUT to take on one of two different values, the influence is only $\log 2 = 1$ bit.

3.3 Equivalence to channel capacity

We now show that that the influence of LOW over $\mathcal{P}$ when HI is hi is exactly the maximum information flow, or *channel capacity*, from LOW to $\mathcal{P}$, given that HI is hi, and that the fact that $\mathtt{HI}=\mathtt{hi}$ is known. Intuitively, the concepts of control and information flow are tightly linked. As we describe in Section 2, dynamic taint analysis has been used both to propagate the flow of control and the flow of information. Hence, it is unsurprising that a quantitative measure of control turns out to be a well known quantitative measure of information flow.

Using Denning's definition of quantitative information flow [12], the amount of information that flows from LOW to OUT, given the known assignment $\mathtt{HI}=\mathtt{hi}$, is the expected reduction of entropy of LOW, given OUT, and given the known assignment of HI:
$\mathcal{I}(\mathtt{LOW}{\rightarrow_{\mathcal{P}}}\mathtt{OUT}|\mathtt{HI}=\mathtt{hi}) =$
$\mathtt{H}(\mathtt{LOW}|\mathtt{HI}=\mathtt{hi}) - \mathtt{H}(\mathtt{LOW}|\mathtt{OUT},\mathtt{HI}=\mathtt{hi}).$

This in turn, is equivalent to the mutual information of LOW and OUT given HI, which can be expressed as $\mathtt{H}(\mathtt{OUT}|\mathtt{HI}=\mathtt{hi}) - \mathtt{H}(\mathtt{OUT}|\mathtt{LOW},\mathtt{HI}=\mathtt{hi})$. Using these relationships, and the fact that the entropy of OUT given LOW and HI is zero (because $\mathcal{P}$ is a deterministic function), we find that the information flow is simply the entropy of OUT, given the assignment of HI:

$$\begin{aligned}&\mathcal{I}(\mathtt{LOW}{\rightarrow_{\mathcal{P}}}\mathtt{OUT}|\mathtt{HI}=\mathtt{hi})\\ &\quad= \mathtt{H}(\mathtt{OUT}|\mathtt{HI}=\mathtt{hi}) - \mathtt{H}(\mathtt{OUT}|\mathtt{LOW},\mathtt{HI}=\mathtt{hi})\\ &\quad= \mathtt{H}(\mathtt{OUT}|\mathtt{HI}=\mathtt{hi})\end{aligned}$$

Channel capacity is the maximum information flow for all probability distributions of LOW [12]. In this case, the probability distribution of LOW that maximizes information flow is the distribution that maximizes the entropy of OUT.

Given the standard definition of entropy of a random variable X as $\mathtt{H}(X) = \sum p(X=x)\log\frac{1}{p(X=x)}$, the entropy of OUT is maximized when all possible values of OUT are equally likely. In that case, the entropy of OUT is equal to the log of the number of values that OUT can take on, which is exactly the influence of LOW over OUT.

Therefore the channel capacity of the information flow from LOW to OUT is exactly equal to the influence of LOW over OUT:

$$\begin{aligned}\max\forall\mu:\mathcal{I}((\mathtt{LOW}\sim\mu){\rightarrow_{\mathcal{P}}}\mathtt{OUT}|\mathtt{HI}=\mathtt{hi})\\ = \mathcal{C}(\mathtt{LOW}{\rightarrow_{\mathcal{P}}}\mathtt{OUT}|\mathtt{HI}=\mathtt{hi})\end{aligned}$$

3.4 Influence variations

We next consider some useful variations of influence.

3.4.1 Partial influence

There are several cases where it is useful to consider influence considering a subset of possible values of LOW, rather than over the entire domain of LOW. The range of $\mathcal{P}$ when considering only a subset of its domain is, of course, a subset of the actual range. Therefore, the influence calculated over a partial domain is a *lower bound* of the actual influence. We refer to the influence calculated over a partial domain as partial influence.

The most obvious reason to use partial influence is in cases where it is difficult or impossible to calculate the influence over the entire domain of LOW, such as some cases where the domain of LOW is of infinite size. We use this technique in Section 6 to reason about influence in large x86 programs.

Partial influence can also be used to reason about a *particular* value of LOW, or a particular *class* of values of LOW. As we have defined it, influence is *independent* of any actual value of LOW; it is a property of the program $\mathcal{P}$ itself. In Section 5.2 we show how partial influence can be used to help classify whether a particular value of LOW exploits an overwrite vulnerability.

3.4.2 Max influence

So far, we have only defined influence as parameterized for a particular value of HI. This formulation is useful when HI is known, *e.g.*, when dynamically determining the influence for a particular execution of the program in question.

For applications where HI is unknown, such as when performing static analysis, it would be useful to consider a variation of influence calculated over *all* possible values of

HI. Usually, it would be most useful to find the *maximum* influence over all values of HI. This is a measure of the *most* control that LOW *could* have over OUT.

4 Influence analysis of problematic program structures

In this section, we examine several program structures that can cause DTADI tools to give incorrect results. For each structure we give simple concrete examples, explanations of when they occur in real programs, explanations of why they are difficult for DTADI tools to handle, and finally we show how the influence measure handles each case. Note that the influence measures stated in this section are calculated by manual inspection. We describe methods for automatically calculating influence in Section 6, and use those methods to analyze these and other programs in Section 7.

In each concrete example, we examine the taint propagation from LOW to OUT, and the influence of LOW over OUT. LOW and OUT are both 8 bit unsigned integers. For simplicity, most of these examples do not depend on HI.

4.1 Sanity checks

We first examine sanity checks implemented by conditional execution: cases where OUT is only derived from LOW after LOW has been found to be within some acceptable range.

Example 1 Sanity check

```
if LOW < 16 then
  OUT:= base + LOW
else
  OUT:= base
end if
```

In Example 1, OUT is derived from LOW only when LOW has been verified to be less than 16. Taint analysis tools do not take such sanity checks into account, and OUT will be considered tainted.[2]

To calculate influence, we observe that for any LOW, OUT can have one of 16 values, from (base+0) to (base+15). Therefore, the influence of LOW over OUT is 4 bits. While LOW has some control over OUT, it does not have a full 8 bits of control over the 8 bit OUT.

We have observed such structures in real programs. In particular, we have verified that the gcc C compiler compiles some switch statements by performing a check on the switch variable, and then using it to calculate an address, which is used in an indirect jump. While the sanity check makes this structure safe, DTADI tools detect this as a security violation when the switch variable is tainted.

As far as we are aware, the only way to work around such cases in current DTADI is to manually annotate the code structure, *e.g.* to force LOW to be untainted after the sanity check, or to ignore the specific integrity violations that are detected as a result of OUT being marked tainted. In addition to requiring manual intervention, care must be taken not to introduce other inaccuracies with such annotations.

4.2 Arithmetic restriction

Example 2 Arithmetic restriction: mask

```
OUT := base + (LOW & 0x0f)
```

A similar problem occurs when arithmetic is used to restrict the range of a calculation. Example 2 is equivalent to Example 1, but instead of restricting what inputs may be used via control-flow, it instead simply masks off the bits that could cause OUT to take on an undesired value. It is possible that some switch structures may be compiled in this way.

All DTADI tools that we are aware of would consider OUT to be tainted in this case, which could lead to false positives. As with Example 1, this problem can be addressed with manual annotations.

The influence of LOW over OUT in this example is again 4 bits, which is expected since the end result of this example is identical to Example 1.

Example 3 Arithmetic restriction: identity functions

```
OUT_1 := LOW XOR LOW
OUT_2 := LOW & 0
OUT_3 := 0; OUT_3 := OUT_3 + LOW; ...; OUT_3 := OUT_3 −
LOW
```

Example 3 shows several examples where, while a naïve analysis may conclude that OUT is derived from LOW, the final value of OUT actually does not depend on LOW at all. In these examples, all three OUT variables have the value 0 at the end of the program. Hence, these kinds of structures can lead to false positives.

DTADI tools that we are aware of detect some of the common structures of this nature, using a precompiled list of idioms. In particular, many compilers exclusive-or a register with itself to initialize it to zero, and all the DTADI tools that we are aware of mark the result of this instruction as untainted.

[2] DTADI tools may only consider OUT tainted when LOW passes the sanity check, since otherwise the derivation from LOW does not take place.

However, in the general case, detecting such cases is non-trivial. Especially consider $\texttt{OUT}_3$ in the example above. In this case, examining either assignment in isolation makes it seem that $\texttt{OUT}_3$ is derived from LOW. It is only by examining the higher-level structure of the program that we can determine that the final value of $\texttt{OUT}_3$ does not depend on LOW at all. We further examine this challenge in Section 6.1.

The influence of LOW over each OUT variable is 0 bits, since each OUT can take on exactly 1 value, regardless of the value of LOW.

4.3 Table lookup

There are several ways in which low integrity inputs can be used to *select* other data. One of the most ubiquitous ways is via a table lookup.

Example 4 Table lookup

```
OUT:= table[LOW]
```

In Example 4, LOW is used as an index into a table. We assume for the sake of this example that the contents of this table are untainted. DTADI tools that we are aware of *optionally* mark OUT tainted in this case, using a rule that data loaded via a tainted pointer is also tainted. As we demonstrate in Section 7.1.3, enabling this rule can cause taint analysis to have false positives, and disabling this rule can cause taint analysis to have false negatives.

The actual influence depends on the contents of the table, which is not shown. For example, if each entry in the table is unique, then OUT can take on 256 unique values, and the influence is 8 bits. Otherwise, the influence is less than 8 bits. In the extreme case where every table entry contains the same value, the influence is 0 bits. In programs where the contents of the table are not constant, the table can be considered to be part of HI.

This case occurs frequently in real programs. This technique is often used to translate input from one character set to another. We have observed functions such as `toupper` and `tolower` implemented in this way. We have also observed this technique used to select a request-handling function pointer in samba (see Section 7.1.3).

4.4 Implicit flows

Another way that tainted data can be used to select untainted data is via control flow.

In Example 5, the final value of OUT is equal to LOW (assuming LOW is 8 bits), yet there was never a direct assignment from LOW to OUT. In the information flow and taint analysis literature, this is referred to as an implicit flow.

Example 5 Implicit flow

```
if LOW == 0 then
  OUT:= 0
else if LOW == 1 then
  ...
else if LOW == 255 then
  OUT:= 255
end if
```

The influence of LOW over OUT in this example is 8 bits, since OUT can take on 256 unique values.

DTADI tools do not track implicit flows, and would consider OUT to be untainted, which could lead to false negatives. Some taint analysis tools that are used for *confidentiality* rather than *integrity* [14, 17] detect *positive* implicit flows, which are implicit flows that result from an *executed* assignment that is control-dependent on LOW. However, none that we are aware of detect *negative* implicit flows, which are implicit flows that result from an *unexecuted* assignment that is control-dependent on LOW, without the assistance of manual program annotations.

To clarify, suppose the program is executed with LOW set to 4. In that case, the assignment (OUT:= 4) is a positive implicit flow, and all the other assignments are negative implicit flows. A bound calculated over the positive implicit flows would approximate that the previous 4 untaken branches could have each leaked at most 1 bit of LOW. However, because of the negative implicit flows (the other branches that are never reached) the influence over, and information leakage to, OUT is actually all 8 bits of LOW.

These types of structures can be used to translate input from one format to another. In particular, keyboard input is propagated via implicit flows in the Windows keyboard driver [6].

5 Using influence to identify attacks

As we have shown in Section 4, influence is a useful theoretical tool for reasoning about how much control LOW has over OUT. However, in the context of detecting integrity violations, there are two questions that must be addressed: how much influence signifies an integrity violation, and how do we use influence to determine whether a *particular* value of LOW is malicious?

5.1 Quantitative integrity policies

As we discussed in sections 1 and 4, current DTADI systems propagate a binary taint attribute at a per-instruction granularity. Ad-hoc rules are used to specify which inputs of a particular instruction have enough control over each output of that instruction to justify propagating the

taint attribute. Using influence, it is possible to specify an integrity policy based on a quantitative threshold of how much influence low-integrity inputs may have over given high-integrity operations, rather than based on a collection of taint-propagation rules. Due to their simplicity, we believe that quantitative policies will be easier to create and understand. Additionally, sensitivity of such policies can be tuned by simply adjusting a numeric threshold rather than attempting to modify or create new qualitative taint propagation rules.

As an example, we consider one of the types of attacks that DTADI systems are used to detect: when a low-integrity input overwrites a function pointer or return address. The policy used in current systems is specified roughly as "data *derived* from low-integrity inputs should not be loaded into the program counter," where *derived* is defined by the collection of taint-propagation rules. Using influence, we can specify a quantitative policy such as "low-integrity inputs should not have more than t bits of influence over a value loaded into the program counter," where t is a simple numeric threshold.

Selecting a threshold t for this policy is fairly straightforward. It is common for untrusted inputs to be able to specify a function pointer from a small set of valid values (*e.g.*, handlers for different request types), so setting t to zero would certainly result in false positives. To our knowledge, though, situations where a low integrity input is able to legitimately choose from more than a handful of pointers are rare. Hence, setting t to a small value such as 6 bits (allowing the low integrity input to select from up to $2^6 = 64$ different choices), would likely result in few false positives. In theory, any non-zero value for t could result in false negatives if a vulnerability allowed the input to overwrite only a few bits of the target pointer. However, we are unaware of any such vulnerabilities, and such a vulnerability would be relatively challenging to exploit in a useful way.

Another advantage of using a threshold-based integrity policy is that different thresholds may be set for different high integrity operations. For example, while it is common for a low integrity input to legitimately have a few bits of influence over a function pointer, a low integrity input should never have any influence over a return address.

5.2 Identifying malicious values of low-integrity inputs

Influence is a property of a program, not a property of a particular input value. Hence, in the general case, influence can be used to identify a vulnerability, but does not indicate whether a particular value `low` of `LOW` is an attack.

One way to use influence to determine whether a particular value of `LOW` is an attack is to calculate the influence over `OUT` for some *class* of value of `LOW`, to which `low` belongs. This is done by calculating the the influence for a partial domain of `LOW`, as described in Section 3.4.1.

For example, the security violation in most overwrite attacks is that a calculated pointer used in a memory-store operation points to an unintended destination; *e.g.* outside of the intended buffer in the case of a buffer overflow. Leveraging this observation, we can determine whether the value `LOW` results in a buffer overflow by calculating the influence over `OUT` for the set of values of `LOW` that would cause all memory store operations to write to the same address as `low` does. When `low` is a value that results in a buffer overflow, this influence measure will be high. When `low` is a value that does not result in a buffer overflow, inputs that write outside of the intended buffer will not be considered in the influence calculation, and the calculated influence will be low (or zero).

6 Measuring influence in binary programs

We next investigate how to automatically measure influence in real binary programs. We show that while it is possible to leverage previous work in taint analysis and information flow analysis, all previous approaches we are aware of are forced to over-approximate, and in practice require manual annotation to calculate useful bounds. We propose a new approach, which is the first that can calculate a sound lower-bound of influence. It can also calculate an upper-bound that is sound for certain cases and useful in practice, and in some cases can soundly calculate the exact influence.

6.1 Transfer-functions are forced to over-approximate

Recall again that influence is defined as the log of the size of the range of a program $\mathcal{P}$, for some particular assignment to high-integrity inputs `HI`. We briefly discuss several possible approaches for calculating influence, using Example 6 as a motivating example. In this example, `OUT` can only take on one possible value: 0x0. As a result, the influence of `LOW` over `OUT` is 0 bits.

Example 6 Inter-dependent operands

```
if predicate(LOW) then
   a := 0x01; b := 0x10
else
   a := 0x10; b := 0x01
end if
OUT:= a & b
```

We first consider the approach of using *transfer functions*, which is the approach of all previous mechanisms of which we are aware for calculating taint or information flow. In these approaches, a transfer function is defined for

each statement in the language, which defines how execution of the statement affects the tracked attribute for each variable. One way of utilizing this approach, which is the most analogous to previous approaches for quantitative information flow tracking, is to track the number of different values that each variable can take on. In Example 6, analysis of the if-then-else structure would determine that a and b can each take on two different values. With only this knowledge, a transfer-function of the final assignment to OUT is forced to over-approximate that OUT could take on $2*2 = 4$ different values, which would in turn over-approximate the influence as 2 bits.

Greater accuracy can be achieved by tracking the actual set of values that each variable can take on, rather than only the size of the set. One implementation of such an approach is value-set analysis [2], which is a method for calculating memory-alias relationships in binary programs. Unfortunately, this approach is still forced to over-approximate. In this example, analysis of the if-then-else structure would determine that a and b can each take on the values 0x01 and 0x10. A transfer-function analysis of the final assignment would conclude that OUT can take on three different values: 0x01 & 0x01 = 0x01, 0x10 & 0x10 = 0x10, and 0x10 & 0x01 = 0x0. As a result, this approach would over-approximate the influence as 1.6 bits.

Example 3 in Section 4.2, in which LOW is added to a variable and later subtracted again, is another example of this type of problem. A transfer-function that analyzed each operation independently would conclude that the influence is non-zero, when in fact it is zero.

Because of these types of problems, transfer-function-based approaches are not able to compute a sound lower bound of information flow, and in practice often require manual annotations to compute a useful upper bound.

6.2 Our approach: end-to-end analysis

To avoid over-approximation, we analyze the end-to-end derivation of OUT as a whole, rather than analyzing each individual step along the way. Our high-level strategy is to soundly convert some or all of the program to a formula, and then use a decision procedure to reason about the range of OUT, given the particular assignment of HI. For programs with a finite number of possible execution paths, and given enough computation time, this approach can theoretically compute the *exact* influence with guaranteed accuracy. The challenge to this approach is to calculate useful bounds of the influence on real programs, in a reasonable amount of time.

6.2.1 Converting the program to a formula

The first step is to generate a formula that accurately models the program, and which a decision procedure can reason about. We here briefly describe the techniques we use to generate such formulas. The details of these techniques are described in our previous work [1, 5, 19].

At a high level, we first convert some or all of the IA-32 binary program to an intermediate representation (IR) with a smaller instruction set, which makes all side-effects (such as setting and checking condition flags) explicit. If necessary, we convert the program to be loop-free by unrolling loops up to a maximum of some fixed number of times. Finally we compute the weakest precondition [13] over the program. Where we are unable to model the entire behavior of the program, guards are inserted to ensure soundness, effectively constraining the domain of LOW to those values which the formula can reason about with guaranteed accuracy.

Generating a formula that accurately models an entire binary program can be quite challenging. Indirect jumps make it difficult to statically determine all possible control flow paths. Modeling loops that could execute an unbounded number of times requires finding loop invariants, which is an open research problem. While system calls can be modeled statically, using a provided HI to specify auxiliary inputs, doing so requires significant implementation effort.

As in our previous work [5, 19], we address these problems by only modeling the execution paths we *are* able to analyze, and using guards to reject formula inputs that would execute an unmodeled execution path. For large programs, we currently model exactly *one* execution path- the path taken in an actual execution of the program. We leverage an execution trace obtained from the TEMU dynamic taint analysis tool [1, 14, 27] to greatly simplify the problem. The execution trace contains the address of each executed instruction, the instruction itself, the values of each operand of each instruction, and the taint attribute of each operand of each instruction.

The resulting formula accurately relates the program inputs to the program outputs, for inputs that would follow the same execution path. Thus, the formula domain is limited to such inputs, and contains guards to reject inputs outside of this domain. The formula is of the desired form: $\mathcal{P}(\texttt{LOW}, \texttt{HI}) = \texttt{OUT}$. HI is obtained from the execution trace. LOW corresponds to inputs that were marked tainted in the execution trace. OUT may be an intermediate or final value of any program state, the value of program output, or any combination.

6.2.2 Determine how many values OUT may take on

The next step is to find how many values OUT may take on. We are not aware of any existing tools that can solve this problem directly. We have developed some strategies for reasoning about how many values OUT may take on using

current decision procedures.

We first give the formula to the decision procedure. We then pose queries to the decision procedure, which consist of predicates over the free variables; in our case `LOW` and `OUT`. The decision procedure responds to these queries either that the predicate is *valid*, indicating that it holds for all values of `LOW` and `OUT`, or responds with a counterexample: an assignment to `LOW` and `OUT` that causes the predicate to be false.

Given this interface, we have developed the following query strategies:

- **Ask for examples of `OUT`.** The most straight-forward technique is to simply query the decision procedure for a value of `OUT` that satisfies the formula, then query for another value of `OUT` that satisfies the formula and has not already been found, continuing to query the decision procedure either until no more values of `OUT` can be found (which will take many queries for high influence values), or until we have found enough values to establish that the influence is higher than some given threshold.
- **Positive range queries.** We expect that values that `OUT` may take on will often occur in contiguous ranges. Once we have found a value that `OUT` may take on, we can perform binary searches to establish the left and right boundaries of such a range. Queries posed to the decision procedure will be of the form "All values from proposed-left-boundary to proposed-right-boundary of `OUT` are satisfiable." Each boundary can be found using a number of such queries less than the log of the size of the co-domain. Note that this type of query requires a universal quantifier, which is unsupported by many decision procedures.
- **Negative range queries.** Similarly, once we have established a value of `OUT` may *not* take on, we can perform a binary search to find the left and right boundaries of the encompassing contiguous range of values that `OUT` may not take on. Unlike the positive range query approach, this approach does not require universal quantifiers, allowing it to be used on more decision procedures.
- **Partitioning.** We can reason about parts of `OUT` independently to help establish an upper bound of influence. For example, if `OUT` is 32-bits, we can establish whether each individual bit can take on values of both 0 or 1 in at most 64 queries. The influence of `OUT` as a whole can be no more than the number of bits that can take on both 0 and 1 values. Tighter bounds can be achieved at greater execution cost by using larger granularities, such as analyzing each byte instead of each bit.
- **Random sampling.** Another approach is to choose random values of `OUT`, and query the decision procedure whether it is possible to satisfy the formula with each of those values. We can then use the fraction of these values that are satisfiable to estimate what fraction of values in the whole codomain of $\mathcal{P}$ are satisfiable. This technique is useful when a significant fraction of the codomain of $\mathcal{P}$ is satisfiable; *i.e.* when the influence is large. For example, when analyzing a potential overwrite attack, if the attacker has 30 bits of influence over a 32 bit `OUT`, 25% of the codomain is satisfiable, and only a few samples are needed for a statistically significant result. However, when the influence is only a few bits, too many random samples would be needed to achieve a tight influence estimate, though we could still use this approach to set a probabilistic upper bound of influence.

We currently implement the strategies of asking for examples, and negative range queries. We first attempt to get up to some threshold t number of example values of `OUT`. If the decision procedure is not able to find any more examples after finding x examples, then we are done, and the influence is exactly $\log x$ with respect to the formula, and *at least* $\log x$ with respect to the program. In cases where the formula is complete; *i.e.* models all relevant parts of the program; then the influence is exactly $\log x$ with respect to the program.

If we find t examples, we stop asking for individual examples. At this point, we have established that the influence is *at least* $\log t$ both with respect to the formula and with respect to the actual program. We next compute an upper bound using the negative range query strategy to find the lowest value `OUT` can take on v_l, and the highest value `OUT` can take on v_h. We then know that the influence is no more than $\log(v_h - v_l)$.

7 Evaluation

We have implemented the method of calculating influence described in Section 6. We currently query for up to 64 values of `OUT`, meaning that we find the exact influence when the influence is up to 6 bits ($6 = \log 64$), and otherwise establish a sound lower bound of 6 bits of influence.

We first apply it to example programs from Section 4. Each of these programs was written as a C program and compiled using gcc. For these experiments we used 32-bit variables for `LOW` and `OUT` instead of 8-bit variables. We then use the tool on larger constructs known to cause problems for DTADI systems: a `switch` statement, and a case-conversion using the `toupper` function. Finally, we demonstrate the scalability and usefulness of our approach on two real programs: samba, and Microsoft SQL server. Using our influence measurement, we are able to verify that an alarm generated by the TEMU [1, 14, 27] DTADI system

alarm in samba is a false positive, and that an alarm in SQL server is a true positive.

7.1 Results

We summarize our results in Table 1. For each experiment we list the program or program-construct evaluated, and how many execution paths are accounted for in the formula. Where possible, we construct a formula including all paths. In these cases, both the lower and upper influence bounds are sound. In other cases, we model only the execution path taken in a given execution trace, as described in Section 6. We next list the time spent performing decision-procedure queries, the measured exact influence or influence bounds, and the lowest and highest value that `OUT` may take on. Finally, as a point of comparison, we list the actual influence as established by manual inspection.

7.1.1 Baseline experiments

The first two experiments are provided as a baseline reference. In the "No propagation" experiment, `OUT` does not depend on `LOW` at all. In the "Direct copy" experiment, `LOW` is simply copied to `OUT`. Our measured results are as expected: the influence is found to be exactly 0 in the first case, and between 6 and 32 in the latter case.

The next group of experiments are analogous to the examples from Section 4. In the table lookup test, each of the four bytes of `LOW` are masked with 0x0f and used as an index into a table with values 0x10 to 0x1f. For each of these experiments, we are able to calculate the exact influence or useful bounds of the influence within a few seconds.

7.1.2 Sanity checked data: switch statements

The `gcc` compiler, and probably many other compilers, compiles some `switch` statements to compute a pointer from the `switch` variable, and then use that pointer as an indirect jump target. Sanity checks ensure that this operation is safe, but this results in a false positive for taint analysis tools when the table lookup policy is enabled and the `switch` variable is tainted. We confirm that TEMU raises an alarm for this case. However, our influence measurement verifies that `LOW`'s actual influence over the indirect jump target is only a few bits.

7.1.3 Solving the taint analysis table lookup catch-22

As we described in Section 4, whether data loaded via a tainted pointer should be marked tainted is a configurable policy in most DTADI tools. In real programs, this typically happens when tainted data is used as an index into an untainted table. We next demonstrate that enabling the table-lookup policy can cause DTADI tools to have false positives, that disabling the table-lookup policy can cause them to have false negatives, and that the influence measurement can be used to accurately reason about these problematic cases.

Table-lookups are often used to convert data from one character set to another. As a result, *disabling* the table lookup propagation policy in DTADI tools can lead to false negatives: tainted data translated in such a way is marked as untainted. We evaluate the gnu libc `toupper` function as an example of such a translation. In this experiment, we translate each of the four bytes of `LOW` via `toupper`, and evaluate the taintedness and influence of the result. When collecting the execution trace for this experiment, we provide the input "aaaa". As expected, when the table-lookup policy is disabled, TEMU marks the result as untainted. Our influence measurement confirms that the input has at least 6.1 bits of influence over the resulting value, and could have up to 28.6 bits of influence of the resulting value for the execution path examined. The actual influence is higher, due to the unanalyzed execution path where the input bytes are not lower case characters. Nonetheless, the measured influence bounds are sufficient to show that the result is heavily influenced by `LOW`.

Table-lookups can also be used to select from a small set of values, by restricting the index to a small set of values, or by having duplicate entries in the table. As a result, *enabling* the table lookup propagation policy in DTADI tools can lead to false positives. In particular, we have observed this to be the case in samba, which we evaluate in Section 7.1.4.

7.1.4 Real-world programs

We next use our influence measurement tool on two real-world programs.

Samba is an open-source implementation of the Windows SMB protocol. Samba uses data from a network request to calculate an index into a table of function pointers, which causes a false positive in TEMU when that function pointer is called. Using our influence measurement tool, we are able to verify that the sanity checks in the program make this operation safe: the input has only a few bits of influence over the function pointer. As a further verification step, we verified that all of the values that the function pointer can take on are within the code segment of the program.

Microsoft SQL server is a closed-source database, and was the target of the well known Blaster worm. Our tool verifies that the alarm generated by TEMU when SQL server is attacked by the Blaster exploit is a true positive. Specifically, we find that the input has at least 6 bits of influence over a return address, and could have as much as a full 32 bits of influence over the return address: *i.e.*, total control. As far as we are aware, the true influence is indeed

Program	Paths Analyzed	Run Time (s)	Measured Influence (bits)	Measured Value bounds	Actual Influence (bits)
No propagation	all (1)	.07	0.0	0x0 to 0x0	0
Direct copy	all (1)	1.1	6.0 to 32.0	0x0 to 0xffffffff	32
Sanity checked (Ex. 1)	all (2)	.54	4.0	0x0 to 0xf	4
Masked (Ex. 2)	all (1)	.28	4.0	0x0 to 0xf	4
Plus-minus (Ex. 3)	all (1)	.23	0.0	0x0 to 0x0	0
Table lookup (Ex. 4)	all (1)	5.5	6.1 to 27.9	0x10101010 to 0x1f1f1f1f	16
Implicit flow (Ex. 5)	all (6)	.57	2.8	0x0 to 0x6	2.8
switch (ijmp)	1 (of 1)	1.9	4.2	0x401078 to 0x4010de	4.2
toupper	1 (of 2)	94	6.1 to 28.6	0x41414141 to 0x5a5a5a5a	31.4
samba (fn ptr table)	1 (of many)	73.6	3.3	0x807de90 to 0x8088ea0	3.3
SQL server (ret addr)	1 (of many)	17346	6.0 to 32.0	0x0 to 0xffffffff	~ 32

Table 1. Influence measurement results.

32 bits.

8 Discussion and future work

8.1 Accounting for unknown HI

In Section 3.3, we showed that influence of LOW over OUT is equal to the maximum information flow, or channel capacity, from LOW to OUT, *when HI is known*. This assumes that the provider of the untrusted input LOW (the potential attacker), knows the complete state of the rest of the program. This is a pessimistic assumption.

In cases where HI is unknown, the influence of LOW over OUT does not change; $\mathcal{P}$ still has the same range. However, the information flow from LOW to OUT may be *less* than the influence of LOW over OUT. In practical terms, this comes about when LOW can influence OUT to take on different values, but some uncertainty remains about *which* value OUT will take on for a given value of LOW.

A notable real-world example is the use of *pointer encryption* [8]. The idea in pointer encryption mechanisms is to store pointers in an encrypted form, and decrypt them just before dereferencing them. Hence, an attacker who is able to overwrite the encrypted pointer cannot predict the value of the unencrypted pointer, generally causing the program to crash when the pointer is unencrypted and dereferenced. If an overwrite vulnerability exists that allows an attacker to overwrite a 32 bit encrypted pointer, then the influence of LOW over OUT will be 32 bits. However, without some knowledge of the encryption key (which is part of HI), the maximum information flow from LOW to OUT is 0 bits.

In general, we believe that the difference between influence and channel capacity of LOW over OUT when HI has some uncertainty can be considered a measure of the *unpredictability* of LOW's influence over OUT. Further work is needed to investigate this relationship.

8.2 Improving performance and accuracy of DTADI systems

As we showed in Section 4, the inaccuracies of DTADI systems stem largely from manually generated, ad-hoc taint propagation policies, defined over one instruction at a time. We believe that these problems can be addressed by using the techniques presented in this work to *automatically* generate taint propagation policies, based on *influence*. In other words, we may be able to combine the advantages of our current end-to-end calculation approach with the performance advantages of the transfer-function-based approach described in Section 6.1.

The premise of this approach is to identify blocks of code, such as a function, that have a single entry point and single exit point. For each such block of code, the *maximum influence* (See Section 3.4.2) of each variable read could be computed for each variable written. When the DTADI system is about to enter one of these pre-computed code blocks at run-time, the summary can be used to determine what the taint value of each written variable should be when the block exits.

This approach could improve the accuracy of current DTADI systems, since interactions within the block, such as sanity checks, implicit flows, and arithmetic interactions, would be accounted for. If desired, this approach could also be used to propagate a quantitative or stratified taint attribute instead of a binary taint attribute, which would help keep track of cumulative effects between blocks, and allow for quantitative integrity policies (Section 5.1).

This approach can also improve the performance of DTADI systems, since the taint propagation summary for a block may be less expensive to execute than propagating a taint attribute for each instruction within the block.

There are, however, several challenges that must be addressed before this approach can be implemented. Some of these challenges, and potential solutions are:

- The code of the target program must be analyzed statically. Code that cannot be found or analyzed statically,

such as dynamically generated code, must still be instrumented at run-time (though these summaries can of course be saved for subsequent program runs).

- For our current influence-calculation techniques to work, we must be able to transform the code block to an equivalent loop-free program, and we must be able to resolve the set of possible targets of indirect jumps. An imperfect analysis may suffice though, using run-time guards to detect if a loop executes more times than the static analysis unrolled it, or if an indirect jump transfers control to an unaccounted-for destination.

- Execution of the block must be atomic, or equivalent to atomic. For example, suppose a pre-emptive thread scheduler transfers control to a different thread while a block is executing. The taint status of each variable is at that point not in a well-defined state. When control returns to the first thread, any assumptions made in the static analysis may have been invalidated. Note that this is only a problem if the threads share data; otherwise the block execution can be modeled as atomic. Some approaches to addressing this problem are to either *force* atomicity, *e.g.* by disabling interrupts, or to implement a recovery mechanism for when an unexpected control transfer occurs.

- We do not statically know which inputs to the block will have been influenced by `LOW`, and we need to account for all possible values of the uninfluenced inputs. Depending on the desired accuracy of the influence calculation, *e.g.* if we want a sound lower or upper bound, it is unclear how to statically account for all of the possible combinations in a scalable way.

Despite these challenges, we believe that this approach is a promising direction for improving the accuracy and performance of DTADI systems.

9 Conclusion

In this work we have proposed *influence* as a quantitative measure of how much control one variable has over another. We have shown that influence is a useful theoretical tool for reasoning about control. We have proposed and implemented an end-to-end technique for measuring influence in binary programs. We have used our influence measurement technique on a number of synthetic and real-world programs, demonstrating its usefulness and practicality.

We believe that influence can be used to greatly improve the state of using dynamic taint analysis to enforce data integrity policies. As a theoretical tool, it can be used to help reason about policies. As a practical tool, we have demonstrated that influence can be used to perform a deeper analysis of generated alerts, thus vetting false positives. We believe that our techniques can also be extended to supplement or replace the manually written taint propagation rules used in today's DTADI systems with automatically generated propagation rules that improve the system's performance and accuracy.

References

[1] BitBlaze research group. `http://bitblaze.cs.berkeley.edu/`.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23. Springer-Verilag, 2004.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, The Mitre Corporation, Mar. 1973.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, The Mitre Corporation, Apr. 1977.

[5] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of USENIX Security Symposium*, Aug. 2007.

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

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

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, 2003.

[9] J. R. Crandall and F. T. Chong. Minos: Architectural support for software security through control data integrity. In *Proceedings of the International Symposium on Microarchitecture*, December 2004.

[10] S. H. D. Clark and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *Proceedings of Quantitative Aspects of Programming Languages*, 2001.

[11] S. H. D. Clark and P. Malacaria. Quantified interference for a while language. In *Proceedings of Quantitative Aspects of Programming Languages*, 2004.

[12] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, Inc., 1982.

[13] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[14] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.

[15] I. James W. Gray. Toward a mathematical foundation for information flow security. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, 1991.

[16] P. Malacaria. Assessing security threats of looping constructs. In *Symposium on Principles of Programming Languages (POPL 2007)*, 2007.

[17] S. McCamant and M. D. Ernst. Quantitative information-flow tracking for C and related languages. Technical Report MIT-CSAIL-TR-2006-076, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, November 17, 2006.

[18] J. K. Millen. Covert channel capacity. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[19] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the* 13^{th} *ACM Conference on Computer and and Communications Security (CCS)*, Oct. 2006.

[20] 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 2005)*, Feb. 2005.

[21] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proceedings of the First European Conference on Systems (EuroSys2006)*, Apr. 2006.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[23] G. E. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, Oct. 2004.

[24] 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*, Dec. 2004.

[25] V. Venkatakrishnan, W. Xu, D. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *8th Internation Conference on Information and Communications Security (ICICS '06)*, Dec. 2006.

[26] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[27] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, Oct. 2007.