Measuring Channel Capacity to Distinguish Undue Influence

James Newsome
Bosch Research and Technology Center
James.Newsome@us.bosch.com

Stephen McCamant and Dawn Song
University of California, Berkeley
{smcc,dawnsong}@cs.berkeley.edu

Abstract

The *channel capacity* of a program is a quantitative measure of the amount of control that the inputs to a program have over its outputs. Because it corresponds to worst-case assumptions about the probability distribution over those inputs, it is particularly appropriate for security applications where the inputs are under the control of an adversary. We introduce a family of complementary techniques for measuring channel capacity automatically using a decision procedure (SAT or #SAT solver), which give either exact or narrow probabilistic bounds.

We then apply these techniques to the problem of analyzing false positives produced by dynamic taint analysis used to detect control-flow hijacking in commodity software. Dynamic taint analysis is based on the principle that an attacker should not be able to control values such as function pointers and return addresses, but it uses a simple binary approximation of control that commonly leads to both false positive and false negative errors. Based on channel capacity, we propose a more refined quantitative measure of *influence*, which can effectively distinguish between true attacks and false positives. We use a practical implementation of our influence measuring techniques, integrated with a dynamic taint analysis operating on x86 binaries, to classify tainting warnings produced by vulnerable network servers, such as those attacked by the Blaster and SQL Slammer worms. Influence measurement correctly distinguishes real attacks from tainting false positives, a task that would otherwise need to be done manually.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]; D.2.5 [*Testing and Debugging*]; E.4 [*Coding and Information Theory*]

General Terms Languages, Measurement, Security, Verification

PLAS '09 June 15, Dublin, Ireland.

Keywords channel capacity, quantitative information flow, model counting

1. Introduction

Dynamic taint analysis is a practical and popular [11, 30, 10, 25, 26] technique for detecting violations of data integrity in computer systems, including commercial software that is only available in binary form. Tainting implements a simple Biba "low water-mark" integrity model [2] with two levels: tainted data (untrusted, low integrity) should not influence sensitive (trusted, high integrity) operations. A binary-level tainting policy will immediately catch the broad class of *overwrite attacks* in which a malicious input causes an unexpected memory write with a value under the attacker's control (typically leading to control-flow hijacking and the execution of the attacker's injected code) [25].

Despite these attractive features, dynamic taint analysis suffers from significant sources of both false positive and false negative errors. Dynamic taint analysis suffers from false positives when programs properly sanitize initially untrusted data, since such sanitization removes the danger caused by untrusted data values without removing their taint. Conversely, dynamic taint analysis suffers false negatives when it fails to track implicit flows such as load-address dependencies and effects mediated by control flow. These errors can be reduced by special case rules and manual program annotations, but these increase development cost and introduce more sources of error. These weaknesses stem from the fact that taint is a binary attribute.

We propose the construct of *influence* to capture in more detail the control that input variables have over an output variable. Influence is based on the information-theory concept of channel capacity. *Channel capacity* is a quantitative information-flow measure that represents a maximum flow over all possible probability distributions of input. This measure is particularly appropriate in an integrity context, since the untrusted input is assumed to be under the control of an attacker, so that its distribution is not known a priori. We show that channel capacity naturally generalizes dynamic taint analysis, and gives accurate results in the cases where dynamic taint analysis is inaccurate.

To evaluate this measure, we present a practical approach for measuring influence, and implement it. Our approach is

a family of strategies that complement each other, giving results that are either exact or have narrow error bounds. Our tool uses these strategies to measuring influence in commodity binary software. (By comparison, previous scalable approaches provide soundness without any guarantee of precision, and previous precise approaches do not scale to off-the-shelf software.) We then apply this tool to the post-analysis of dynamic taint analysis alerts. Our tool's results confirm some alerts as true positives, and show that others are false positives due to sanitized data.

In summary, this paper makes the following contributions:

- We give a family of complementary techniques for measuring channel capacity with a decision procedure that give precise results for both small and large capacities.
- We show how to implement channel capacity measurement in a tool that runs without source code or human annotations on off-the-shelf x86 binaries.
- We apply channel capacity measurement to the problem of false positives in dynamic taint analysis, showing that it accurately distinguishes real attacks from false positives.

2. Background

We begin by reviewing two concepts from information-flow analysis that will be linked in this work: the channel capacity of a program, and the technique of dynamic taint analysis.

2.1 Channel capacity

In information theory, the capacity of a possibly noisy channel is the maximum amount of information it can transmit. In quantitative information-flow analysis, the same concept is applied to a computer program, for instance to measure the flow from a program input to a program output. If the probability distribution of the inputs to a program is known, a program output can be considered as a random variable whose distribution determines the amount of information about the input it conveys. Specifically, the *entropy* of a random variable X, $\mathtt{H}(X)$, is given by $\sum p(X = x) \log_2 \frac{1}{p(X=x)}$. If the probability distribution over the inputs is unknown, the information flow can be safely approximated by taking the distribution that maximizes the output entropy; this is the *channel capacity* [13]. It is not hard to see that the entropy is maximized when each output is equally probable.

Channel capacity is a good match for the adversarial situations that occur in security, in which the job of a defender is to minimize the damage that an attacker could do, given that the attacker tries to do maximum damage. It would usually be unsafe to assume that the attacker's inputs are chosen from a fixed (for instance, uniform) distribution, since the attacker might use knowledge of the program to choose only the most dangerous values. Instead, channel capacity represents an upper bound that applies regardless of the strategy an attacker uses.

2.2 Dynamic taint analysis

Dynamic taint analysis is a flexible technique that can be applied to programs at the source or binary level to track the movement of security-sensitive data. Though it is sometimes used to track secret data, we concentrate on the more common case of an *integrity* policy that tracks untrusted data and ensures it does not influence security-sensitive operations. For instance, dynamic taint analysis can be used to automatically detect code injection attacks against servers by tainting untrusted network inputs and preventing tainted instructions from being executed. This overview is brief; interested readers may refer to an extensive literature [11, 30, 10, 25, 26] for more implementation details and practical evaluation.

The principle of dynamic taint analysis is to associate some extra metadata with each value a program operates on (for instance, each variable in a source code analysis or each register and memory location in a binary analysis). This tainting information records that a particular value is untrusted, and optionally other information like the original source of the untrusted information. The key feature of this taint attribute is that it *propagates* to later values that are computed based on a tainted value: for instance, a copy of a tainted value is tainted. An operation, say addition, that takes both tainted and untainted values as input produces a tainted value at its result. Then, to check whether a data value used in a sensitive operation (like a control flow transfer) is trustworthy, a dynamic taint analysis must simply check whether it is tainted.

Limitations of dynamic taint analysis. Unfortunately, dynamic taint analysis suffers from two generic kinds of inaccuracies. First, because taint propagates through any direct flow, a data value may still be tainted after a program has verified that it is in fact safe. Unless such a value is explicitly untainted (say via a manually written annotation, impractical for the analysis of off-the-shelf binaries we target), it can produce a false positive report. Conversely, however, tainting generally does not propagate via more indirect kinds of program relationships like the addresses used for memory accesses and the previous control flow decisions that caused execution to reach a certain point. However, these indirect relationships, collectively called *implicit flows*, can still have an effect on computations performed later in a program. If a value fails to be tainted because it was only indirectly affected by the untrusted input, a false negative (missing report) is the result.

The limitations of dynamic taint analysis stem from the fact that tainting is only a binary attribute: a value is either tainted or it is not. To more precisely distinguish safe from unsafe programs, we would like tainting to be a matter of degree, and to know what values an attacker might produce. To address these limitations, we introduce a more graduated

attribute of *influence*, which is an application of channel capacity.

3. Influence for accurate dynamic taint analysis

In order to improve on the results of dynamic taint analysis, we introduce influence as a more finely gradated attribute for a data value, based on channel capacity. This section first introduces influence in general, then shows how it applies in situations where simple tainting is inaccurate. We then discuss possible approaches for building a quantitative integrity policy based on influence.

3.1 Influence

To measure the damage that an attacker can do to the control flow integrity of a program (as a refinement of dynamic taint analysis), we specialize channel capacity to what we call *quantitative influence*, or just "influence."

In what follows, we restrict ourselves to programs that are deterministic from the perspective of the attacker, and we assume that any inputs not under the attacker's control are fixed for the purposes of analysis. In this case, both the influence and the channel capacity are given by the logarithm of the number of possible program outputs.

In more detail, we consider a deterministic computation $\mathcal{P}(\mathtt{I}, \mathtt{I}_{aux}) \rightarrow \mathtt{V}$. The inputs to the computation are partitioned into I and $\mathtt{I}_{aux}$, where I are the possibly malicious inputs we are interested in (e.g. data read from the network), and $\mathtt{I}_{aux}$ is the set of all other inputs to the computation (e.g. data read from a configuration file). Non-deterministic computations could be modeled as taking a random variable as part of the auxiliary input $\mathtt{I}_{aux}$, but we concentrate on deterministic ones. Moreover, we fix the effect of the auxiliary input entirely by taking a particular value $\mathtt{I}_{aux} = \mathtt{c}$.

Definition The *feasible value set* of V is the set of all values that could be assigned to V by the computation $\mathcal{P}$.

Definition The *(quantitative) influence* $\mathcal{I}$ of I over V is the $\log_2$ of the size of the feasible value set.

Note that in general, the influence of I over V is a more accurate characterization than the number of bits that I can affect in V. For example, if we have the program "`if (p(I)) V = 0x0 else V = 0xffffffff`", for any predicate p, the number of bits of V affected by I is 32 bits—the same as if I had been directly assigned to V. However, the influence from I to V is only 1 bit.

For the application we consider, the value of the influence will be the same as the channel capacity, but that equality may not hold in more complicated situations with non-determinism or other inputs that are neither fixed nor completely under the attacker's control. We also find that the term "influence" more accurately conveys the intuition that we are interested in measuring the degree of control that an attacker has over the program's execution.

3.2 Influence in action

In this section we describe several program structures that are known to be problematic for dynamic taint analysis tools. For each program structure we explain why dynamic taint analysis does not accurately characterize the influence that I has over V, while the influence measurement does. We also discuss when each problem occurs in real-world programs.

In each example, we examine the influence of the untrusted message I over the final value of the variable V. For simplicity, I and V are 8 bit unsigned integers, and the examples do not take auxiliary inputs $\mathtt{I}_{aux}$.

We start with examples that cause false positives from dynamic taint analysis, then discuss ones that cause false negatives.

3.2.1 False positive examples

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

Example 1 Sanity check

```
if I < 16 then
  V:= base + I
else
  V:= base
end if
```

In Example 1, dynamic taint analysis tools will find that V is tainted by I in cases where the sanity check passes. This is misleading, because in fact I has very limited influence over V due to the sanity check.

In contrast, the feasible value set is $[\text{base}, \text{base} + 15]$, and the quantitative influence is 4 bits. Both of these measures provide a more accurate characterization of the influence that I has over V.

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

Example 2 Arithmetic restriction: mask

```
V := base + (I & 0x0f)
```

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

As in the previous example, dynamic taint analysis tools measure the influence as tainted, while the feasible value set is $[\text{base}, \text{base} + 15]$, and the quantitative influence is 4 bits.

All dynamic taint analysis tools that we are aware of would consider V to be tainted in this case, which could lead to false positives.

3.2.2 False negative examples

There are several ways in which data from an untrusted message can be used to *select* other data. One of the most ubiquitous ways is via a table lookup.

Example 3 Table lookup

```
V:= table[I]
```

In Example 3, I is used as an index into a table. Dynamic taint analysis tools that we are aware of *optionally* measure V as tainted in this case, using a rule that data loaded via a tainted pointer is also tainted. Disabling this rule can lead to false negatives in cases such as when untrusted data is translated from one character-set to another via a lookup table, which we have observed in functions such as `toupper` and `tolower`. Conversely, enabling this rule without some way of accounting for data sanitization can lead to false positives when untrusted data is used to select from a table of pointers. We investigate a false positive of this nature in Samba in Section 5.3.

The quantitative influence in this example is the logarithm of the number of unique values in the table.

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

Example 4 Implicit flow via branching

```
if I == 0 then
  V:= 0
else if I == 1 then
  ...
else if I == 255 then
  V:= 255
end if
```

In Example 4, the final value of V is equal to I (assuming I is 8 bits), yet there was never a direct assignment from I to V.

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

Some dynamic taint analysis tools do not track implicit flows, and would consider V to be untainted, which could lead to false negatives. Some dynamic taint analysis tools that are used for *confidentiality* rather than *integrity* [22, 16] do track implicit flows. Note that dynamic taint analyses for data integrity do not track implicit flows not only because of the additional complexity required to do so, but because most implicit flows yield relatively little influence.

These code structures can be used to translate input from one format to another. For instance, keyboard input is propagated via implicit flows in the Windows keyboard driver [6].

3.3 Influence-based integrity policies

Dynamic taint analysis may be used, for instance, to implement a policy of the form "data derived from the network may not be loaded into the program counter" to prevent control-flow hijacking by remote attackers. Using quantitative influence, a corresponding policy would be "data influenced x or more bits by the network may not be loaded into the program counter". Our primary contribution is a measurement technique that could be used with an arbitrary policy, but here we briefly discuss the policies in which such measurements can be used to detect overwrite attacks.

In an *overwrite attack*, a maliciously crafted input causes a memory write operation to overwrite an unintended destination with a value controlled by the attacker. The more detailed information produced by our influence analysis can catch overwrite attacks by highlighting two features: the amount of influence the attacker has is high, and the specific values he can produce are unsafe.

Thresholds for influence. In a control-flow hijacking overwrite attack, the unintended destination is one that is later loaded into the program counter, or used to derive a value that is later loaded into the program counter. Note that by our definition, an overwrite attack results in a write to an unintended destination, of a value substantially controlled by the attacker (typically part of the attacker's input). In such cases, the attacker's influence over a 32 bit value loaded into the program counter will be nearly 32 bits. It may be not quite 32 bits due to constraints imposed by the protocol; e.g. inputs with white-space or null characters may be rejected by the program.

In more general situations, there is a set of program counter values that could allow an attack (e.g., pointing to an area that could contain attack code); an attack is possible if this set intersects the feasible value set. The larger the feasible value set, the more likely it is that this intersection will be nonempty. However, a small feasible value set is not on its own a guarantee that no attack is possible, for instance if an attacker is able to fill most of a program's memory with copies of malicious code.

Thus, we expect that in general the influence of an untrusted input over the program counter will be nearly 32 bits in the case of a control-flow hijacking attack, and nearly 0 bits otherwise, with cases in between being relatively rare. In our experiments in Sections 5.2 and 5.3, we found that the largest legitimate influence was 4.17 bits, while the smallest attack influence was much larger, about 26 bits. Thus there is wide scope for a policy to distinguish attacks from false positives.

Unsafe feasible values. In the influence measurement tool that we developed, we explicitly identify values that are in

the feasible value set. In cases where we have information about what values of V are acceptable, we can use this information directly to implement policies such as "untrusted inputs should not be able to cause the program counter to take on illegal values". For example, when V is a value that is loaded into the program counter, values that would result in a segmentation fault, such as zero, can be assumed to be unacceptable. In some cases more refined information about safe and unsafe values may be available; for instance, we might check for a library code address that could be used for a return-into-libc attack, or an area of memory that is known to contain network input.

4. Techniques for measuring quantitative influence

We have designed an approach for measuring quantitative influence (channel capacity), and implemented it in a tool for binary programs. The approach is based on an accurate model of program behavior, which allows it to obtain precise quantitative results. Unlike type-system or static-analysis approaches, which use abstraction and generally guarantee a one-sided soundness property but not precision, our technique gives results that are exact or have a tight probabilistic error bound. The high level idea of our approach is to build a sound model of the program in a first-order logic. We then use a decision procedure to reason about the feasible value set, which can then in turn be used to reason about the quantitative influence.

The model we build is a predicate over I, $\mathtt{I}_{aux}$, and V. Because the model is *sound*, for any assignment to I, $\mathtt{I}_{aux}$, and V for which the predicate is satisfied, the program is guaranteed to compute the corresponding value for V when given those values for I and $\mathtt{I}_{aux}$.

For small, loop-free programs, the model is also *complete*. That is, for any assignment to I, $\mathtt{I}_{aux}$, and V such that the program computes V when given I and $\mathtt{I}_{aux}$, the predicate is guaranteed to be satisfied. For programs that are too large or complex to model completely, including most real-world programs, we restrict the model to a particular execution path, as described in Section 4.2.2.

We employ several query strategies using a decision procedure to reason about the set of values of V for which the predicate can be satisfied. In cases where the model is complete, this is the feasible value set, and the quantitative influence is the base-2 logarithm of the size of that set. In cases where the model is not complete, this is a *subset* of the feasible value set, and a *lower bound* of the quantitative influence is the log of the size of that set.

We now discuss the details of the design of our technique for quantitative influence measurement, and its implementation in a practical tool.

4.1 Overview of solution

As input, our technique takes: (1) a loop-free binary program (or an execution trace rewritten as a loop-free program as described in Section 4.2.2); (2) a variable V, specified as a storage location (e.g. CPU register or memory location) and program point (instruction pointer); (3) a description of which storage locations or system calls correspond to the untrusted input I over which to measure influence; and (4) concrete values c for auxiliary inputs $\mathtt{I}_{aux}$.

As output, it produces: (1) what we can soundly and probabilistically determine about the feasible value set; (2) sound low and high bounds of the quantitative influence; and (3) a probabilistic estimate of the quantitative influence.

At a high level, our approach is as follows:

- Step 1: We convert the program to a logic predicate by disassembling it, converting to an intermediate representation, and then calculating the weakest precondition [14] over the program. The result is a model of the program in quantifier-free first-order logic. For large programs, we perform this conversion for a single path, but in any case no abstraction is performed: the model is bit-precise.
- Step 2: We pose a series of queries to a decision procedure to learn about the feasible value set.
- Step 3: We calculate the quantitative influence from the feasible value set.

We now describe these steps in more detail.

4.2 Step 1: Program to predicate

We generate a predicate that accurately models the program, and which a decision procedure can reason about. At a high level, we first convert 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. We then compute the weakest precondition [14] over the program. The details of these techniques are described in our previous work [3, 29, 24, 4].

4.2.1 Static program conversion

For relatively small programs, up to a few functions, our technique can accurately compute the complete influence of an input variable on an output variable. We statically extract all the instructions in a program from a 32-bit ELF binary, and convert any initialized data areas (such as tables) into explicit initialization statements. Our tool can automatically inline function calls and unroll loops up to a supplied bound to obtain a loop-free program. We used this static conversion for the illustrative examples of Section 5.1.

4.2.2 Converting a single path

For larger programs, complete (all-paths) influence becomes impractical to compute precisely, and often is not even meaningful. (For instance, if the set of possible program outputs is unbounded, the channel capacity is infinite.) So

we apply our approach to large programs by computing the influence that is possible along a single control-flow path. In this section, we first motivate this single-path approach in more detail, then describe how it is implemented.

Uses for a single path. The feasible value set with respect to executions on a single path is a *subset* of the feasible value set with respect to the entire space of program executions. Similarly, the quantitative influence over a path is less than or equal to the quantitative influence over the entire space. Hence, the influence with respect to a path can be interpreted as a *sound lower bound* of the whole-program influence.

There are two important ways in which the influence over a single path may be significantly less than the total influence. One is that there may be significant flows on a path that was not analyzed, such as in the program "if (`I` $<$ 1000) `V = 0`; else `V = I`". The second possible situation is that a program may have many paths, but the influence on every path individually is small; for instance, this is the case in Example 4. Despite the potential for underestimating influence due to these issues, we show in Section 5.2 that for real-world exploits the calculated influence is sufficiently high to be confident that they are indeed exploits.

Measurement on a single path. To measure influence along a single, observed, path we obtain an execution trace from the TEMU dynamic taint analysis tool [3, 16, 34], and restrict our model to the execution path followed in that trace.

The execution trace contains the address of each executed instruction, the instruction itself, and the values of each operand. As in our previous work [24, 4], we convert this execution trace to a straight-line program that accurately models the corresponding execution path in the original program. We add guards to ensure that we reject inputs that would follow unmodelled execution paths.

The influence calculated in this modified program corresponds to the influence from inputs that would have followed the same execution path and used the same memory-write destinations as logged in the trace. The feasible value set is thus a subset of the feasible value set for the entire program, and the quantitative influence is a lower bound. While this could result in false negatives when classifying exploits, we did not find this to be the case in our evaluation of real-world exploits in Section 5.2.

4.3 Step 2: Predicate to feasible value set

Given the predicate that specifies the relationship between `I` and `V`, our tool calculates the influence by a series of queries about the model to a decision procedure. This is similar to a game of "Twenty Questions," but in addition to yes or no answers, the decision procedure can give a satisfying assignment if our query involves free variables. The goal is to obtain enough information to get an accurate influence bound without requiring too many queries (for instance, asking separately about the feasibility of each output point would be much too slow).

For illustration, we visualize the feasible value set as a number line representing the potential range of `V`. For instance, in our experiments, `V` is a 32-bit value, so this number line goes from 0 to $(2^{32} - 1)$. Initially, the feasibility of the entire number line is *unknown*. As we pose queries to the decision procedure, we learn points and ranges on the number line that are *feasible* or *infeasible*. Feasible points are values that `V` can take on, for some value of `I`. Infeasible points are values that `V` cannot take on, no matter what value of `I` is chosen.

As a final step, we also perform sampling over any remaining unknown space in the value set, giving us a probabilistic estimate of what fraction of that space is feasible. As an alternative probabilistic technique we also use an approximate model counting approach.

The decision procedure we use is STP [17]. STP takes as input a predicate in a quantifier-free first order logic over bitvectors (which can represent machine integers) and arrays, and outputs whether or not that predicate is satisfiable. If it is satisfiable, it also outputs an assignment to the variables in the predicate that allows the predicate to be satisfied.

As building blocks, we use the following basic query strategies:

- Point feasibility query: Check whether a particular value is feasible or not.
- Range feasibility query: For some range, check whether a feasible value exists in that range or not. If so, the decision procedure gives us an example feasible value. If not, we learn that the entire range is infeasible.
- Exhaust-up-to-c: Repeat the range feasibility query up to some number of times c, or until there are no more feasible values.
- Density sampling: Choose points at random within some range, and use point feasibility queries to determine whether or not they are feasible. Use this to estimate the fraction of that range that is feasible.
- Probabilistic model counting: Using the "XOR streamlining" approximation to #SAT [18], we add k random parity constraints to the model, and check whether the augmented model is still feasible. Each random constraint reduces the number of feasible outputs by a factor of 2, so if the augmented model has at least one feasible output, the original probably has at least 2^k.

These techniques have complementary strengths. Exhaustive output querying gives exact results when the number of feasible outputs is small, and negative range results provide a sound upper bound that is tight when the feasible outputs are clustered. Conversely, density sampling gives a probabilistic estimate that is accurate to within a fraction of a bit when a large fraction of the outputs are feasible. Proba-

bilistic #SAT is relatively expensive, but its accuracy (within 1 or 2 bits in our experiments) is sufficient for many applications, and it is equally applicable whatever the number of feasible outputs.

In our experiments, we use the following combined query strategy:

- Use a range feasibility query to get any feasible value. (There must be at least one, barring implementation errors).
- Use point feasibility queries to check whether the smallest (0) and largest ($2^{32} - 1$) values are feasible.
- If either of those were infeasible, use a series of range feasibility queries as a binary search to "widen" the infeasible point to the next feasible point. After this, we have established the lowest and highest feasible values.
- Repeatedly query remaining unknown ranges to either mark them as infeasible, or divide them at a feasible point. We repeat this step until the space is exhausted, or we have found c (64 in our implementation) more feasible points. Thus every bound of up to $\log_2(64) = 6$ bits our tool computes is exact.
- Perform density sampling over any remaining unknown space, and compute a 95% confidence interval using a likelihood ratio technique [15].

As we show in our evaluation, this query strategy allows us to learn quite a bit about the feasible value set in a reasonable amount of time for real-world programs.

In examples where exhaust-up-to-c does not already give an exact bound, we also separately test probabilistic #SAT, using a simple iterative approximation algorithm. Our tool keeps a running estimate of the influence, at each iteration does an XOR-streamlining experiment based on that estimate, and increases or decreases the estimate based on the results.

4.4 Step 3: Feasible value set to quantitative influence

During the query process, our tool maintains its knowledge about the feasible value set in the form of an exhaustive list of ranges, each of which is either completely feasible, completely infeasible, or unknown. It computes the influence by summing the total number of feasible points and then taking the logarithm. Definitely feasible ranges are always counted, and definitely infeasible ranges are never counted. For a lower bound, unknown regions are counted as if they were infeasible. For an upper bound, unknown regions are counted as if they were feasible. For a probabilistic bound, unknown regions are counted in proportion to the probabilistically sampled density.

5. Evaluation

We next demonstrate our influence measurement techniques on a number of examples. Our experiments show that feasible value set and quantitative influence are useful measures: they are more accurate and give a better understanding than dynamic taint analysis using a binary taint attribute. Our experiments also show that our techniques to measure quantitative influence provide useful bounds and estimates of these measures in a reasonable amount of time.

```
int implicit(int input)
{
  int output = 0;
  if      (input == 0) output = 0;
  else if (input == 1) output = 1;
  /* ... */
  else if (input == 6) output = 6;
  else output = 0;
  return output;
}
```

Figure 1. Implicit flow

```
int popcnt(unsigned int i) {
  i = (i & 0x55555555) + ((i >> 1) & 0x55555555);
  i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
  i = (i & 0x0f0f0f0f) + ((i >> 4) & 0x0f0f0f0f);
  i = (i & 0x00ff00ff) + ((i >> 8) & 0x00ff00ff);
  return (i + (i >> 16)) & 0xffff;
}
```

Figure 2. Population count

```
unsigned int mix_copy(unsigned int x) {
  unsigned y = ((x >> 16) ^ x) & 0xfff;
  return y | (y << 16);
}
```

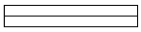
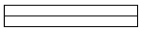
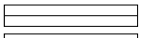
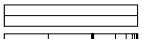
Figure 3. Mix and duplicate

In Section 5.1, we measure the feasible value set and quantitative influence for several small, illustrative examples. We use these examples to illustrate what the influence measure means in real programs, and to develop an understanding of the output that our tool produces.

In Sections 5.2 and 5.3, we use our tool to evaluate alarms generated by the TEMU dynamic taint analysis tool in *real-world* programs. Our results show that we are able to reason about influence in real-world programs well enough to soundly confirm alarms as true positives (Section 5.2), and to identify other alarms as likely false positives (Section 5.3).

5.1 Illustrative examples

Table 1 shows our tool's results on several illustrative examples. The purpose of these experiments is primarily to demonstrate the feasible value set and quantitative influence measurements for easy-to-understand programs. These examples also show that while we do not obtain the exact feasible value set and quantitative influence measures when the

Program	Feasible Value Set	Quant Influence (bits) Low Bnd	High Bnd	Sample	#SAT
Copy (`v = i`)		6.04	32.0	[31.8, 32.0]	32.0
Masked copy (`v = i & 0x0f`)		4.00	4.00	N/A	N/A
Checked copy (Example 1)		4.00	4.00	N/A	N/A
Divide by 2 (`v = i / 2`)		6.58	31.0	[30.8, 31.0]	31.7
Multiply by 2 (`v = i * 2`)		6.58	32.0	[30.4, 31.6]	31.5
Implicit flow (Figure 1)		2.81	2.81	N/A	N/A
Population count (Figure 2)		5.04	5.04	N/A	N/A
Mix and duplicate (Figure 3)		6.04	32.0	[0.0, 28.6]	15.8

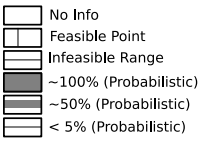


Table 1. Influence measures for example programs. The feasible value sets are summarized graphically as show in the legend at right.

influence is large, our tool still presents enough information about these measures to be useful.

Each program was written in C and compiled before being given to our tool. For each example, we use the function parameter `input` as `I`, and the value returned via `eax` as `V`. The influence is measured over the *entire* domain of 0 to $2^{32} - 1$. That is, we generate a model that is complete with respect to the computation; it accounts for all possible execution paths.

For each program, we obtain information about the feasible value set and quantitative influence of `I` over `V`. We show the feasible value set graphically. For the quantitative influence, we show the sound low and high bounds, and (when the exact bounds do not already match) a probabilistic estimate based on sampling (as a confidence interval) or approximate #SAT. To show separately the performance of the different techniques, the results in each column are independent; our tool's actual reported estimates are combination of the different results as described in Section 4.4.

In the `copy` function, `I` is simply copied to `V`, and hence `I` can cause `V` to take on any value from 0 to $2^{32} - 1$, giving it 32 bits of influence over `V`.

In the `masked_copy` (`output = input & 0x0f`) and `checked_copy` functions (Example 1), `I` is copied to `V`, but `V` is restricted to values between 0 and 15. In the `masked_copy` function this is done by masking off all but the low 4 bits of `I`, while in the `checked_copy` function this is done by only executing the copy when `I` is between 0 and 15. In both cases, `I` has 4 bits of influence over `V`.

In the `div` and `mult` functions, `V` is set to `I` divided by two (unsigned), or multiplied by two, respectively. This gives `I` control over all but the most significant bit, or all but the least significant bit, respectively. Hence, in either case, the influence is 31 bits.

The example `implicit` is an implicit flow. There is no direct assignment from `I` to `V`, but `I` is able to "choose" between seven values for `V`, giving it $\log_2 7 \approx 2.8$ bits of influence over `V`.

The remaining two examples demonstrate characteristics of a computation that present difficulty for some analysis techniques. The population count example (Figure 2) counts the number of bits in a word that are 1. Some outputs of this function are much less frequent than others, so randomly choosing inputs would be very unlikely to, for instance, find the one input that causes the output 0. But because our approach samples the output space instead, it can easily find all 33 possible outputs. The mix and duplication example (Figure 3) combines the two halves of its input word with XOR, and then duplicates these 16 bits in both the upper and lower halves of its output, so its influence is 16 bits. But this influence is too large to be effectively measured exhaustively, too small for effective sampling, and too uniformly distributed for range queries, so only our probabilistic #SAT strategy gives an accurate estimate.

In each case, our tool finds sound low and high bounds of the quantitative influence. Recall from Section 4.3 that to establish a lower bound of n, we must find 2^n values in the feasible value set, which requires 2^n STP queries. In these experiments, we establish a lower bound of up to 6 bits; if we have not exhausted all feasible values at that point, we perform sampling to establish a probabilistic estimate. In each experiment where the influence is less than 6 bits, we find the exact influence. In those where the influence is greater than 6 bits, one of the probabilistic estimates (sampling or #SAT) is correct within 0.2 bits. The exhaustive and sampling measurements were obtained in 0.5 to 3.8 seconds, while the probabilistic #SAT measurement was slightly slower, taking up to 30 seconds for 200 queries.

5.2 Confirms attacks

In the next set of experiments, we measure the feasible value set and quantitative influence in several real-world vulnerable programs. As described in Section 4.2.2, we scale our techniques to real-world programs by using an execution trace, and measuring influence over the inputs that would follow exactly that execution path. The potential drawback to this approach is that the influence over this path could under-represent the influence over the whole space of executions; e.g., `I` may be able to exert greater influence over `V` when other possible execution paths are considered.

Therefore, these experiments are designed to evaluate whether we are able to scale our techniques to real-world

Program	Time (hrs)	Feasible Value Set	Quant Influence (bits) Low Bnd	High Bnd	Sample	#SAT	#SAT Time
RPC DCOM (Blaster vuln)	6.0		6.07	32.0	[31.8, 32.0]	30.4	2.7 hr
SQL Server (Slammer vuln)	0.23		6.36	30.9	[26.7, 28.3]	26.6	12 min
ATPhttpd	8.8		6.04	32.0	[31.8, 32.0]	31.0	4.5 hr

Table 2. Influence measures for known attacks in real-world programs.

```
int do_switch(char *src)
{
  int i=0;
  switch (src[0]) {
  case 'a': i = 1; break;
  case 'b': i = 2; break;
    /* ... */
  case 'r': i = 18; break;
  default: i = 19; break;
  }
  return i;
}
```

Figure 4. Switch on input

programs using this trace-driven approach, and whether the influence measured with respect to a single path is sufficient to identify attacks.

In each experiment, TEMU detected that data tainted by a network message had been loaded into the program counter. We measure the influence of the network message, which ranges from 376 bytes for SQL Server to 1822 bytes for RPC DCOM, over the 32-bit program counter at the point where the violation was detected. The experiments ran on a 3.0GHz Core 2 Duo E8400 with 6MB of cache and 4GB of RAM. (Though many of our query techniques could be parallelized, our single-threaded prototype used only one core).

Our results are presented in Table 2. The measurements took from 14 minutes to about 9 hours. In each of these examples, the influence is measured sufficiently to identify each of these as actual attacks. For a non-attack, we suspect that it is rare for a network message to have more than a few bits of influence over the program counter—even the lower bound established in these measurements is enough to strongly incriminate these examples as true positives. Likely most accurate are the sampling and #SAT estimates, which range from 26 to 32 bits, all very high. Finally, with some knowledge of what values are valid, we can use the feasible value set as further evidence. For example, in RPC DCOM and ATPhttpd, the untrusted input is able to cause the return address to take on the value 0x0, which is obviously invalid.

These results show that our measurement technique can obtain useful influence measures on the order of hours. These measures can be used to verify alarms generated by a dynamic taint analysis tool as true positives.

5.3 Reveals over-tainting

The next question we sought to answer is whether our influence measurement tool can provide a better understanding of how an untrusted input influences a variable in cases where dynamic taint analysis has false positives; e.g., where sanity checks restrict how much influence the untrusted input has over the variable in question.

To answer this question, we examine several cases where TEMU raises alarms for non-attack messages (false positives). As in the previous experiment, in each case we measure the influence of a network input, restricted to follow the same execution path as our execution trace, over a 32-bit variable. We summarize our results in Table 3.

The first experiment is from the Windows RPC DCOM service. In the previous experiment where we analyzed an attack against this service, we noticed that the `esp` register was also being marked tainted. In our experience this is unusual, and could potentially signify a vulnerability. However, using the influence measure, we see that the input actually has very little control over the value of `esp` at the program point we analyzed. Further manual analysis revealed that this is caused by a stack allocated array, where the size of the array is calculated from the message `I`, as in `char a[f(I)]`.

The second experiment is in the Samba file server. TEMU raises an alert for many requests, due to the program counter becoming tainted. Again, our influence measure shows that the untrusted input actually has very restricted control over the program counter at the program point where TEMU raises the alert. Further manual investigation reveals that the alert is caused by the program using a field from the network message to calculate an index into an array of function pointers, which is used to select which function to call to handle the rest of the request. The calculation is done in such a way as to ensure that the final function pointer selected is one of a few predetermined values.

The third experiment is a synthetic program (Figure 4), in which a byte read from the network is used in a C `switch` statement. Dynamic taint analysis can have a false positive in such cases, if the compiler constructs a jump table and calculates a pointer from the `switch` variable to be used in an indirect jump. In this experiment, we found this to be the case for some `switch` statements, using the `gcc` compiler. As expected, TEMU raises an alarm when tainted data is used as an indirect jump target, but the influence measure shows that the pointer in question can only be chosen from a small set of values, and manual inspection of the compiled

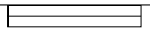
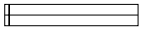
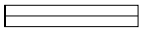
Program	Time (s)	Feasible Value Set	Quant Influence (bits) Low Bnd	High Bnd
RPC DCOM esp (dyn mem alloc)	4.3		3.81	3.81
Samba (fn ptr table)	240		3.32	3.32
synthetic switch stmt (Figure 4)	5.3		4.17	4.17

Table 3. Influence measures for over-tainted variables.

code confirms that `gcc` puts the proper checks in place such that the operation is safe.

The results of these experiments show that the influence measure is a useful tool for helping to understand cases where an untrusted input has restricted control over a sensitive variable, and can be used to help identify and understand false positives in dynamic taint analysis.

6. Discussion and future work

Some directions for extension are suggested by the limits of our current experimental results.

6.1 Limitations and extensions

Our approach is relatively slow for real-world programs, even when considering only one execution path. As such, it is mostly useful in a post-analysis step when using a less accurate influence measurement tool such as dynamic taint analysis. It is also useful as a "gold standard" by which to measure the accuracy of future tools that sacrifice some accuracy for better performance.

Our tool has the option of considering all execution paths, which only works for small programs with relatively few execution paths, or one execution path, obtained from an execution log. When we consider only one execution path, the influence over that path may be less than the influence over other inputs. A useful extension to our tool would be to incrementally consider more execution paths (such as by inverting some branches or generalizing over loops [28]).

6.2 More detailed statistics

Our probabilistic sampling and approximate #SAT query techniques are currently fairly simple, but they could be further enhanced by applying standard statistical techniques. For instance, our current method for obtaining a numeric estimate based on the results of the XOR streamlining experiments could likely be improved to extract more precise results from the same number of iterations.

6.3 New query techniques

The query techniques our tool currently uses treat the output as a single integer, but other approaches would likely work better for larger outputs with internal structure such as strings and arrays. One such strategy is partitioning the input into pieces and then analyzing each separately. For example, to calculate the exact influence over a 100 byte variable exhaustively would require up to $2^{8\times100}$ decision procedure queries. But by partitioning the variable first, the same method would use only $2^8 \times 100$ decision procedure queries. However, if done naively, this strategy may be quite imprecise: the correct value of the influence on the 100 bytes considered together might be anywhere between the maximum of the 100 byte influences and their sum.

Because our decision procedure is based on SAT, it only supports queries without quantifier alternation. Our tool can query whether a range of outputs contains no feasible values, but the seemingly similar query of whether they are *all* feasible is not possible. However, some restrictions can make such queries possible for a SAT solver. Our tool could make positive range queries if it supplied a hypothesis about the input/output relation on a range, such as that the output is a linear function of the input. For instance, in the Blaster example of Section 5.2, the output word is a copy of a certain input word; if it could guess this hypothesis, our tool could verify that the influence is exactly 32 bits in only one query.

Another direction for expanding the queries our tool can perform would be to use a more powerful decision procedure. A decision procedure that took arbitrarily quantified boolean formulas (QBF) would allow positive general positive range queries, and an exact boolean model counting (#SAT) procedure could give an exact influence result in one query. However, these kinds of decisions procedures are less mature than the SAT-solver-based one we now use, and they lie in even higher complexity classes.

6.4 Compositional analysis

Since our technique gives precise results quickly for small pieces of code, approaches that use our technique in a targeted way could achieve precision and scalability simultaneously. For instance, our approach might be used to incrementally refine inaccuracies in a dynamic taint analysis, by heuristically locating code fragments that might cause false positives. Or, our approach might be applied independently to each small unit (e.g., function or basic block) in a program, and then the numeric results combined using a compositional technique such as network flow [22].

7. Related work

Among related work, we first discuss systems based on dynamic tainting, and then other approaches to information-flow analysis.

7.1 Dynamic taint analysis for 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 [11, 12, 30, 10, 25, 26]. While these approaches work well for many programs, they propagate the taint attribute at the instruction level, and round the influence attribute to "tainted" or "untainted" at each instruction. This can lead to false positives and false negatives when multiple instructions interact in complex ways, such as in data sanitization and in implicit flows.

Xu et al. [33] 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.

7.2 Dynamic taint analysis for data confidentiality

Dynamic taint analysis has also been used by several systems to enforce Bell-LaPadula data *confidentiality* policies [6, 34, 16, 22]. These systems are similar to dynamic 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. In practice, the taint propagation policies in these systems are quite similar to those that are targeted to enforce data integrity, though they are often tuned to propagate the taint attribute more aggressively; e.g. the instruction-level propagation policies are tuned to "round up" to marking things tainted. The system proposed by Egele et al. [16] also employs static analysis to account for some ('positive') implicit flows, but does not handle other 'negative' implicit flows without manual annotation.

7.3 Information flow

There is a large body of work on information-flow security. Sabelfeld and Myers provide a good survey of the field [27].

Most prior work seeks to detect or prevent *any* flow of sensitive data to an insecure output. Vachharajani et al. [31] propose and implement a system to dynamically detect unpermitted information flows in binary programs. Venkatakrishnan et al. [32] 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 [13], 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 [23] and by Gray [19].

Several alternative ways of measuring information flow appear in previous work. Perhaps most common is an entropy-based definition using a fixed input distribution, as for instance in the techniques of Clark et al. [7, 8]. The belief-based framework of Clarkson et al. [9] is theoretically appealing, but depends on bounds on the attacker's prior beliefs that may be hard to support. The channel capacity approach we argue for here has seen increasing interest recently, such as in Malacaria and Chen's application of the Lagrange multipliers [21]. McCamant and Ernst [22] also use channel capacity as a goal, though their results bound it only in terms of expectation. Other techniques, such as the precise analysis of loops by Malacaria [20] and the complete enumeration of Backes et al. [1], are flexible enough to be applied to multiple definitions.

Two recent projects scale quantitative upper bound flow measurement techniques to confidentiality policies in binary applications. Castro et al. [5] create privacy-protecting bug reports, then use a conservative approximation of entropy over a uniformly distributed input space to bound the amount of information the report reveals. They generate constraints from a program using a similar technique to our implementation, but use an SMT solver only to find a new test case for a bug: their quantitative measurements do not require solving constraints. Though some of their case studies use software vulnerabilities, their concern is not with the flow that constitutes the vulnerability, but with the private information a bug report could reveal. McCamant and Ernst [22] also use a conservative upper-bound approximation, including manual annotations for implicit flows, but transform to a network flow problem to compute a bound. Like our use of channel capacity, this approach avoids making assumptions about an input probability distribution. Because they represent only an upper bound, the results of these approaches have no guarantee of precision, though they were found to be sufficiently precise for their respective case studies.

In concurrent work, Backes et al. [1] give a measurement technique that exhaustively enumerates a partition of the input space into sets of inputs that produce the same outputs, a similar high-level approach to our exhaust-up-to-c query strategy. However their technique is more sophisticated, using abstraction refinement and lattice point enumeration (model counting for linear constraints) to precisely count the number of inputs that produce each output; this allows many measurements beyond channel capacity as well. Like exhaust-up-to-c, their technique fails to scale to programs with many outputs (their largest example has channel capacity 4.8), such as the attacks we consider in Section 5.2.

8. Conclusion

In this work, we carefully examine a number of program structures, such as data sanitization and implicit flows, that can force dynamic taint analysis into false positives or false negatives. We show that an end-to-end measure of *channel capacity* from an untrusted input to a program variable is an intuitive and accurate characterization of the influence that the untrusted input has over that variable. We devise a family of new techniques for measuring this influence that are precise (exact for small flows, and accurate to within a fraction of a bit for larger ones) and applicable to off-the-shelf x86 binaries. Our tool measures influence in well-known attacks on commodity software, and automatically

distinguishes real attacks from similar executions that cause false positives in a state-of-the-art taint analysis.

Acknowledgments

This research was supported in part by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

[1] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, May 2009.

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

[3] BitBlaze project. `http://bitblaze.cs.berkeley.edu/`.

[4] 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.

[5] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 319–328, Mar. 2008.

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

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

[9] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *18th IEEE Computer Security Foundations Workshop*, pages 31–45, June 2005.

[10] 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.

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

[12] J. R. Crandall, F. T. Chong, and S. F. Wu. Minos: Architectural support for protecting control data. *Transactions on Architecture and Code Optimization (TACO)*, 3(4), Dec. 2006.

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

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

[15] S. Dorai-Raj and S. Graves. Adjusting likelihood ratio confidence intervals for parameters near boundaries applied to the binomial. In *Joint Statistical Meeting (JSM)*, Aug. 2006.

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

[17] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007.

[18] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *21st AAAI Conference on Artificial Intelligence*, 2006.

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

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

[21] P. Malacaria and H. Chen. Lagrange multipliers and maximum information leakage in different observational models. In *Workshop on Programming Languages and Analysis for Security*, pages 135–146, June 2008.

[22] 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*, 2008.

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

[24] 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.

[25] 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)*, Feb. 2005.

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

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

[28] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In

International Symposium on Software Testing and Analysis (ISSTA), July 2009.

[29] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, pages 1–25, Hyderabad, India, Dec. 2008. Keynote invited paper.

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

[31] 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.

[32] 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)*, Dec. 2006.

[33] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, 2006.

[34] 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.