DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation

Min Gyung Kang[*†] Stephen McCamant[†] Pongsin Poosankam[*†] Dawn Song[†]
[†]UC Berkeley [*]Carnegie Mellon University
{mgkang,ppoosank}@cmu.edu {smcc,dawnsong}@cs.berkeley.edu

Abstract

Dynamic taint analysis (DTA) is a powerful technique for, among other things, tracking the flow of sensitive information. However, it is vulnerable to false negative errors caused by implicit flows, situations in which tainted data values affect control flow, which in turn affects other data. We propose DTA++, an enhancement to dynamic taint analysis that additionally propagates taint along a targeted subset of control-flow dependencies. Our technique first diagnoses implicit flows within information-preserving transformations, where they are most likely to cause under-tainting. Then it generates rules to add additional taint only for those control dependencies, avoiding the explosion of tainting that can occur when propagating taint along all control dependencies indiscriminately. We implement DTA++ using the BitBlaze platform for binary analysis, and apply it to off-the-shelf Windows/x86 applications. In a case study of 8 applications such as Microsoft Word, DTA++ efficiently locates just a few implicit flows that could otherwise lead to under-tainting, and resolves them by propagating taint while introducing little over-tainting.

1. Introduction

Dynamic taint analysis (DTA for short) is a popular and powerful technique for tracking information flow in software even without access to source code. DTA works by marking certain inputs to a program as *tainted*, and then *propagating* that taint to other values that are computed transitively based on those tainted inputs. For instance, we can check for information disclosure bugs in a desktop application by marking sensitive inputs as tainted, and then checking whether they propagate to inappropriate outputs (called *sinks*) [8, 16, 37]. However, a significant limitation of standard approaches to DTA is that they do not propagate taint along *control dependencies* (also called *implicit flows*): parts of a program where tainted data values affect control flow, and then the control flow variation affects other data. This can lead to *under-tainting*, a type of error in which values that should be marked as tainted are not, and so for instance could cause an analysis to fail to detect a leak of sensitive information.

Our hypothesis is that under-tainting occurs at just a few places within large benign applications, such as in the implementation of some kinds of data transformations (our results in Section 6 are similar to those seen in some previous studies [21]). To obtain correct tainting results, we would like to fix the under-tainting problems that occur at these locations to propagate taint to the results of such a transformation. A common approach when operating at the source-code level is to use static analysis to find all possible control dependencies, and to propagate taint along each one. However, this approach has two difficulties when applied at the binary level [10, 29, 30]. First, it is difficult to perform precise static analysis on binaries, because they lack many of the structures that were present in source. Second, propagating taint indiscriminately often leads to *over-tainting*, or *taint spread*, the opposite problem from under-tainting that occurs when too many values are tainted.

Thus there is a spectrum that ranges in the extremes from no taint propagation for control dependencies (vanilla dynamic taint analysis) to universal taint propagation for control dependencies (based on static analysis). A useful middle ground is to perform targeted taint propagation for just some control dependencies. Based on the observation that under-tainting usually occurs at just a few locations, we propose to identify targets for additional propagation ahead of time, in an approach we refer to as *DTA++*. In particular, we concentrate on the common case of information-preserving transformations in benign programs. Note that we focus on enabling taint propagation for benign programs only. Malicious programs in which an adversary might intentionally design implicit flows to frustrate analysis [7] are out of scope. Information-preserving transformations, such as the conversion of data from one format to another, occur in many contexts and it is important that they properly propagate taint. Our approach has two phases: first we generate *DTA++ rules* by diagnosing branches responsible for under-tainting and determining the extra propagation they require using offline analysis, and then we apply those rules during future runs of dynamic taint analysis.

Key to our approach is diagnosing only those implicit flows that are likely to cause under-tainting. The intuition behind our diagnosis technique is that if a transformation as a whole is information-preserving, then it may redistribute information between data flow and control flow, but it will not destroy information. In some cases the code may move partial information about a value into control flow, but if there is information remaining in a data value, the data will still be tainted so no under-tainting will occur. The implicit flows that cause under-tainting are the rest: those that transfer *all* of the information about the input into control flow, leaving the data untainted. We can also weaken the assumption of complete information preservation by instead looking for implicit flows that transfer most of the information about the input into control flow. The intuition of the "amount of information transferred to control flow" can be made precise as a kind of quantitative information flow measurement of the branches that have been taken during a program's execution.

Once we have detected such an implicit flow, there are several possible approaches for localizing it: for instance, we could compute a small unsatisfiable core of the branch conditions [9] to find a set of branches that were involved in the implicit flow. However we have found a simpler technique to work well in practice: we use a binary search to find a minimal prefix of the program trace that contains the implicit flow; then the last instruction in this prefix is a branch that is necessary to the implicit flow.

Once the diagnosis technique has identified a branch that could be responsible for under-tainting, our system then generates targeted propagation rules using an instruction-level control-flow graph. (This phase is similar to the propagation that has been proposed in other binary-level techniques (e.g., [10]), with the key difference that we perform it more selectively.) Once we have generated a set of DTA++ rules, our system can apply them on any future dynamic taint analysis runs with just a lightweight modification to an existing DTA tool.

In this paper we present our DTA++ technique, implement it as an enhancement to an existing DTA tool, and evaluate it in a realistic application to tracking sensitive information. We implement DTA++ on top of BitBlaze [6, 31]. In an extended case study, we show how DTA++ obtains correct tainting results in large off-the-shelf applications such as word processors. We also show vanilla DTA often loses tainting because of implicit flows. On the other hand, simply propagating taint for every implicit flow leads to an enormous taint spread (orders of magnitude more tainted bytes in our experiments).

In summary we make the following contributions in this paper:

- First, we propose an efficient and effective technique that identifies a minimum set of implicit flows in the program that potentially cause under-tainting. Given an execution trace with taint information, our technique automatically diagnoses the under-tainting problem if it exists, and then generates targeted taint propagation rules to resolve the under-tainting.
- Second, we implement our technique using the BitBlaze binary analysis platform [31]. Our system uses dynamic and static analysis approaches together to detect under-tainting of a value in a program execution trace, diagnose its cause, and generate propagation rules, and then applies the rules in the course of dynamic taint analysis.
- Lastly, we evaluate our technique by applying it to under-tainting problems that we have encountered in common off-the-shelf word processors on Microsoft Windows. The results of this case study show that our technique accurately identifies the implicit flows involved in under-tainting, and corrects the tainting with few side effects. Our technique also introducing orders of magnitude less taint than when propagating taint for all implicit flows as in previous systems such as DYTAN [10].

The rest of the paper is organized as follows. In Section 2, we describe previous research efforts related to this paper. We define the problem of under-tainting from implicit flows and describe our underlying assumptions in Section 3. We give our approach in Section 4, and Section 5 provides implementation details. We present a case study applying our technique in Section 6 and discuss several issues related to our technique in Section 7. Section 8 concludes this paper.

2. Related Work

Dynamic taint analysis is a popular means for analyzing both benign and malicious software components. Several different techniques have been proposed based on dynamic taint tracking for detecting unknown vulnerabilities in software [11, 12, 26, 34]. They taint potential input sources of malicious data such as network packets; monitor how the tainted input data propagate throughout program execution; and raise an alarm when the taint contaminates sensitive data like return addresses in the stack or user privilege configuration. The main ideas are also similar in analyzing malicious software components leaking sensitive user information on the system [15, 22, 25, 33, 36, 37]. There are also ongoing efforts to apply the dynamic taint analysis techniques to track and confine confidential information in production systems running inside a virtualized environment [16, 17].

However, taint analysis techniques have several challenges in achieving accurate analysis results. Schwartz et al. [29] point out several fundamental challenges including under-tainting and over-tainting. A major cause of under-tainting is implicit flows caused by control dependencies, since vanilla dynamic taint analysis tracks only data dependencies. Implicit flows are especially problematic in applications that require analyzing information flows within malware, because adversarial program authors could potentially embed very complicated implicit flows in their programs to evade analysis. For instance Cavallaro et al. [7] describe these and other challenges in using dynamic taint analysis in a fully adversarial context. In this paper we limit our scope to applying DTA to benign applications, though note that tracking information flows through benign software is valuable in detecting whether that information reaches malicious software. A more pessimistic assessment of the applicability of DTA that includes propagation across memory accesses (as we use in this work) is given by Slowinska and Bos [30]. Our work tackles some of the same challenges they identify, but we argue our results show the challenges are not insurmountable.

The challenges we refer to as under-tainting and implicit flows from control dependencies have been studied since at least the 1970s [14,18], but the lion's share of previous work has been performed on source code, and often requires developer effort such as annotations or refactoring during development. It is much more difficult to deal with such flows in pre-existing binary applications. An example of a recent system that attempts this is Clause et al.'s DYTAN [10]. Our approach is a refinement of one like DYTAN that performs a similar propagation, but it uses a more narrowly targeted selection of branches for which to propagate taint (our diagnosis phase), in order to reduce over-tainting in the results.

In concurrent work, Bao et al. [5] propose what they call "strict control dependence" to include some but not all control dependencies in taint tracking and related analyses. Their definition gives similar results as ours in many cases, but is based on the syntax of a comparison expression. By contrast we use a more general and precise semantic-level condition, implemented using symbolic execution. Bao et al.'s implementation uses a compiler transformation, so it could not be applied to proprietary programs such as many of those in our case study.

3. Problem Definition and Approach Overview

In this section we define in more detail the problem that our DTA++ approach solves. We start by giving some background about under-tainting in general, then give an overview of our approach, and illustrate with examples.

3.1. Background and Motivation

First, we define the concept of under-tainting. Given a (benign) program and a tainted (sensitive) input, we perform taint propagation to see what other values are tainted (contain sensitive information). In taint analysis, under-tainting is a situation where a value is not tainted even though it is affected by the tainted input. We intentionally choose not to make the concept of "affected" completely formal, since which kinds of input-output relations should propagate taint often depends on the details of the analysis intent. A general intuition is that if we compare two executions of a program in which the untainted inputs were the same and the tainted inputs were different, intermediate values and outputs that also differ between the runs are candidates for tainting. However in many applications we do not wish to treat all such differing values as tainted, if the causal relationship between the input and the differing value is too indirect or attenuated.

The under-tainting problems that we want to solve are those caused by implicit flows. We define an implicit flow to be a program structure in which tainted data affects control flow, so that the control flow difference might in turn affect other data. (Schwartz et al. [29] refer to this as *control-flow taint*.) A more general concept of implicit flows would also include structures in which tainted data affects an array index or pointer: for instance, a table lookup when the table index is tainted. Such examples could also naturally be addressed by the technique of this paper, but existing DTA systems already implement a special case for such table lookups which gives the same results, so we have not needed to do so. We call those implicit flows that cause under-tainting *culprit implicit flows*, and the corresponding branch instructions *culprit branches*.

To find culprit implicit flows, we focus on *implicit flows in (nearly) information-preserving transformations*. (We say that a transformation is information-preserving if it implements an injective function: every legal input value produces a distinct output value.) These occur when all or nearly all of the information present in a particular input value affects the program's control flow, so that no data-only taint propagation would occur. Another way of stating this condition is that only a single input value, or only a few, would cause the program to take the same execution path that occurred on an observed run.

3.2. DTA++ Approach Overview

To address the aforementioned problem, we propose DTA++, an enhancement to vanilla dynamic taint analysis (DTA) that propagates additional taint along targeted control dependencies in order to ameliorate under-tainting caused by implicit flows. Given a (benign) program, our goal is to identify data transformations containing culprit implicit flows at which taint should be propagated to avoid under-tainting. For efficiency, DTA++ operates in two stages. First, an offline analysis phase, based on test executions, detects and diagnoses any under-tainting that may be present, and generates DTA++ rules specifying how to propagate taint to eliminate the under-tainting. Second, the online taint propagation phase applies the DTA++ rules during any future use of dynamic taint analysis on the same program, to perform targeted propagation to prevent under-tainting. A graphical overview of this structure is shown in Figure 4.

Offline analysis. The input for the offline analysis phase is one or more execution traces from a program that have been generated using vanilla dynamic taint propagation. A trace may contain under-tainting: that is, there may be a portion of the program input and a corresponding part of the program output such that the input region is tainted, and the output region is derived from the input region via an information-preserving transformation, but the output region in untainted. As output, the analysis phase produces a set of *DTA++ rules*: specifications of additional taint propagations needed to prevent the under-tainting. Each DTA++ rule gives a culprit branch in the program and a list of instructions control-dependent on that branch, such that if the condition at the branch is tainted, the values written by each control-dependent instruction should also be tainted.

The offline analysis is based on test executions which each demonstrate under-tainting of the values derived from some part of program input. Generating such test cases is outside the scope of this paper; we discuss this point further in Section 7.

Online Taint Propagation Using DTA++ rules. Using the propagation rules generated by the offline analysis phase, we can then perform future runs of dynamic taint analysis with simple modifications to apply additional taint according to the DTA++ rules. Since the specification of the DTA++ rules is general, and programs tend to have only a few instances of culprit implicit flows, offline analysis performed on a few test executions of a program will generally suffice to determine how to propagate taint for any future executions.

For this paper, we focus on implicit flows in transformations that are completely information-preserving; thus, our technique will look for implicit flows in which a control-flow path completely determines the value of an input value. In our experience, these are the implicit flows that most commonly lead to under-tainting. Our technique also naturally generalizes to a looser quantitative condition on how close a transformation is to being information-preserving, but we leave as future work how to best set such a threshold to balance false positives and false negatives.

3.3. Examples

Figure 1 shows a simplified example of an implicit flow that causes an under-tainting problem. The C code reads a character from a plain text input and converts it into the Rich Text Format (RTF) [27]. As shown in the code, when a given character is a *control character* like braces (`{` and `}`) and backslash (`\`), it encodes the character to a control code starting with a backslash. For example, `{` in the plain-text input is converted into a two-byte control code `\{`. In the code, the `if` and `else if` clauses in line 4 and 9 have implicit flows that assign to the output the same value as the input value, without directly copying the original input value. Therefore, when we taint a brace character in the input data for dynamic taint analysis, the taint does not propagate to the brace character in the output buffer. Figure 2 illustrates this under-tainting graphically.

The implicit flows at the `if` and `else if` clauses in Figure 1 are typical of those we wish to locate. The transformation converting from plain text to RTF is information-preserving, and for instance the execution path that causes the program to output `\{` can occur only when the corresponding input character is `{`.

By contrast we do not wish to propagate taint for implicit flows of a small fraction of the information in an input value. Such implicit flows occur commonly in large programs, but often the relationship of original input is indirect, so we would not wish to treat all such implicit flows as propagating taint. Empirically, propagating taint for all implicit flows leads to unacceptably-large over-tainting, as demonstrated for instance in our experiments in Section 6. An example of an implicit flow for which we do not wish to propagate taint is shown in Figure 3. In this example, the value `output` contains a small portion of the information contained in the original input: each output value can be caused by many different input values. (Quantitatively, the code transforms the 32-bit integer `input`, with 2^{32} possible values, into one of only two possible values of the buffer `output`, reducing 32 bits of information to only 1.) We expect that for most applications, it would not be desirable to propagate taint for this implicit flow, so we would set the detection threshold to exclude examples such as this.

Thus as a diagnosis solution, we want to locate culprit implicit flows that cause the under-tainting results in the output. Our offline analysis phase processes an instance of the program execution and provides the exact locations in the program code that cause the given under-tainting problem. Since we work on program binaries, the location of an implicit flow is a conditional jump instruction in the binary code. For example, as shown in Figure 7, an implicit flow

```
char output[256];
char input = next_input();
long len = 0;
if (input == ’{’) {
  output[0] = ’\’;
  output[1] = ’{’;
  len = 2;
}
else if (input == ’\’)
  output[0] = ’\’;
  output[1] = ’\’;
  len = 2;
}
/* ... */
else {
  output[0] = input;
  len = 1;
}
add_output(output, len);
```

Figure 1. C code for RTF conversion, with culprit implicit flows

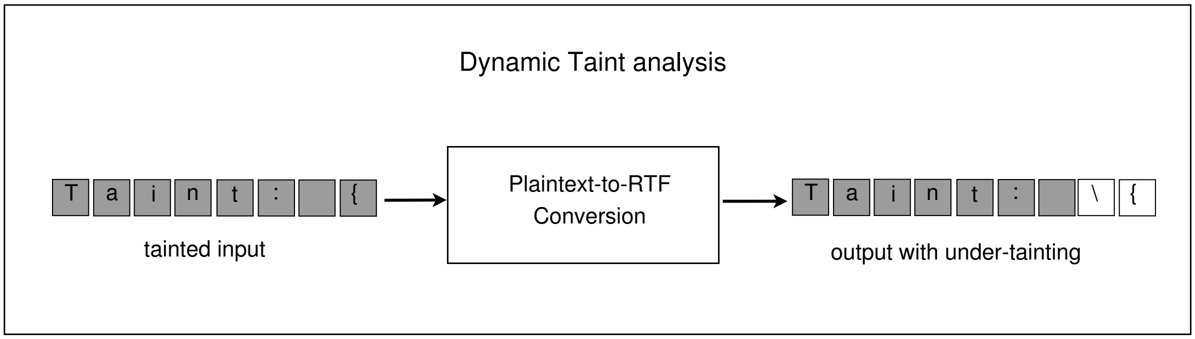

Figure 2. Under-tainting in the plain-text-to-RTF conversion code

causing the under-tainting in `{` (hex value `0x7b`) begins at the jump instruction after `cmp %dl, %al`; and this is the point that we want to automatically find in the execution trace.

4. Our Approach: Offline Analysis

In this section we present the details of our approach to diagnosing and ameliorating the under-tainting problem defined in Section 3. Since applying DTA++ rules requires only a straightforward modification to a standard DTA system, we concentrate on the offline analysis phase that generates those propagation rules. First, we describe how to locate culprit branches from a given execution trace (diagnosis), and then we explain how to create rules to fix the corresponding culprit flows (rule generation). An overview of how these phases work together is shown in Figure 4.

4.1. Diagnosing Under-Tainting

The basic intuition of our diagnosis approach is to search for parts of the execution that make control-flow decisions based on the input that is under-tainted, where the results of those decisions imply tight restrictions on the possible values of that input. Since determining alternative possible inputs requires more fine-grained information than basic taint analysis, we use symbolic execution and base our search on the path predicate that describes a particular execution. In the terminology of symbolic execution, the trace of instruction executions corresponds to a *program path* consisting of a sequence of branch decisions. We refer to any contiguous sub-sequence of instructions from this trace as a *path substring*, and a path substring that starts at the beginning of the trace as a *path prefix*. We describe this search in two parts: first, a detection predicate ϕ to determine whether a path substring has a culprit implicit flow, and then a search procedure to efficiently find the first path prefix for which ϕ holds.

Detection predicate. We implement the detection predicate ϕ using symbolic execution. The principle of symbolic execution is to replace certain values such as program inputs with *symbolic variables*, so that computations produce formulas instead of concrete values. We say that a branch condition is a *symbolic branch condition* if it depends on the symbolic variables. Then the *path predicate* for an execution is the conjunction of the formulas for each symbolic branch condition. Thus the path predicate is a formula over the symbolic variables that holds for executions that take the same control flow path.

$\phi(t)$ will be a predicate that, when given a substring t of an execution trace, returns true if the substring has a cul-

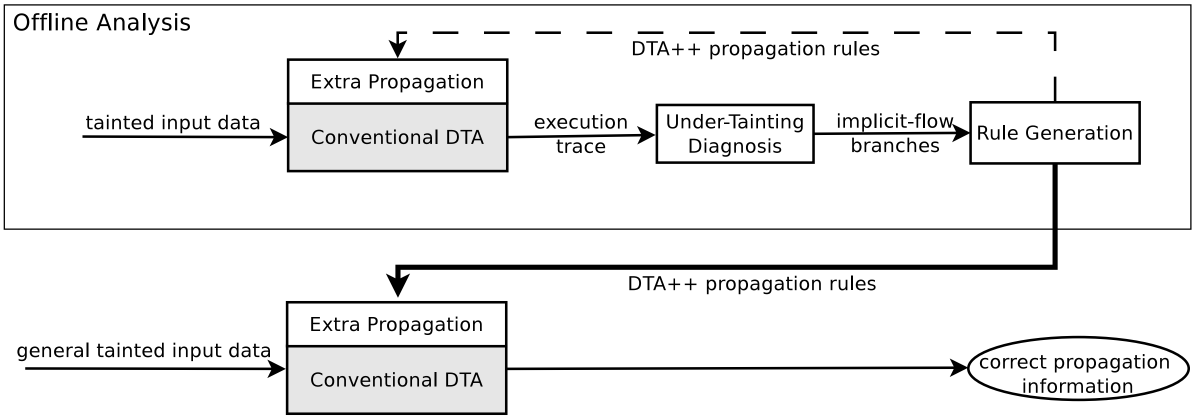

Figure 4. System overview. Our DTA++ system extends conventional dynamic taint analysis (DTA) with under-tainting diagnosis (Section 4.1) and rule generation (Section 4.2) to ameliorate under-tainting caused by implicit flows. Shaded boxes correspond to pre-existing components, while white-background components are the contributions of this work.

```
char output[256];
long input = user_input();
long len = 0;
if (input > 100) {
  strcpy(output, "large");
  len = 5;
}
else {
  strcpy(output, "small");
  len = 5;
}
print_output(output, len);
```

Figure 3. Example code with a non-information-preserving implicit flow. Such implicit flows usually do not cause under-tainting, so we do not want to detect them, and in fact this example is not considered a culprit implicit flow by our system.

prit implicit flow that tightly constrains its inputs. For ease of description, we concentrate here on the highest possible threshold for constraint. This is an implicit flow where the control flow encodes all of the information about the tainted input, or equivalently, when a control-flow path can be reached by only a single input value. Thus we want $\phi(t)$ to be true if there is only one value of the relevant part of the input that causes the program to take the execution path observed in the trace.

To check this condition, we take the parts of the program input that produced the under-tainted value as the symbolic variables. We use symbolic execution to extract a path condition as a formula over that input. Then we query a constraint solver to check whether there is a second solution to the path condition, besides the input values that appeared in the original trace.

For example, when the code in Figure 1 has the input {, the execution path is lines 2, 3, 4, 5, 6, 7, and 19, and the path predicate is `input == '{'`. Then, we attempt to solve this path constraint with the additional constraint that the input be different from the concrete value {: i.e., `input == '{' && input != '{'`. Trivially in this example, the solver tells that there is no other possible value that satisfies the constraint, so the execution path encodes the precise value of the input byte: this is a path for which ϕ holds.

Locating a culprit branch. Using the predicate ϕ, finding the location of a culprit branch reduces to finding the smallest prefix of the execution that satisfies ϕ. (If the whole trace does not satisfy ϕ, then the algorithm reports that it does not have under-tainting.) A brute-force approach would be to try each prefix of the trace starting from the smallest, but this might require $O(n)$ calls to the predicate ϕ, which could be inefficient. Assuming the predicate ϕ must at least do some processing on each instruction in the trace prefix, this would imply at least quadratic complexity. (In fact constraint solving is potentially even more expensive, NP-hard in general, a further incentive to reduce the number of solver calls required.)

We use a more efficient approach based on binary search. We fix the starting point of the trace segment, and use binary search to find the earliest ending point of the trace such

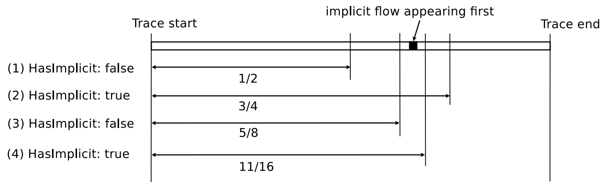

Figure 5. Binary search for a culprit branch in a trace

that the segment contains a culprit implicit flow. Figure 6 shows our algorithm, where the function `HasImplicit` implements the predicate ϕ. Figure 5 illustrates the search in action. Given a trace containing a culprit implicit flow, we split the trace and **(1)** try `HasImplicit()` on the first half of the trace. Because `HasImplicit()` returns `false`, we reduce the search space to the second half of the trace. By splitting the second half of the trace, we **(2)** try `HasImplicit()` on the first three quarters of the trace, and in the example, the result is `true`, limiting the next search space to the third quarter of the trace. Continuing, we try `HasImplicit()` on **(3)** the first 5/8, **(4)** the first 11/16 of the trace, and so on.

Multiple culprit branches in a trace. The trace may contain multiple culprit branches, and we may need to diagnose and propagate taint for several to achieve the desired tainting result. If there are multiple culprit branches, the diagnosis algorithm presented above finds the one that appears earliest in the trace. Once we diagnose one such location, we remove the corresponding formula from the path condition. After this modification, we can perform the diagnosis process a second time to find another culprit branch, and in this way repeat the process until the diagnosis finds no more culprit implicit flows.

4.2. Rule Generation

After diagnosing the cause of under-tainting, we determine how to ameliorate it by generating rules specifying how to propagate taint to the values affected by the flow. To do this, we adopt a technique much like that used by Clause et al. [10]. First, we extract control flow graphs (CFGs) from the program binary and build a database of the immediate post-dominator of each conditional branch by inspecting the CFGs. (A point p in a CFG *post-dominates* a point i if every path from i to the exit passes through p, and *immediately* post-dominates it if there is no other p' for which p' post-dominates i and p post-dominates p'. Immediate post-dominance is a precise counterpart to the intuition of the re-convergence point of a branch, and can be efficiently computed using compiler algorithms [24].)

For each culprit branch our tool diagnoses, we query the database for its immediate post-dominator, and generate a rule specifying to taint the destination operands of all the instructions up to the post-dominator.

The most precise approach to this taint propagation is to taint any register or memory location that might have a different value depending on whether the true or the false side of the culprit branch was taken. Thus, a value written to inside the branch need not be tainted if it is always written the same value on both sides of the branch. Conversely, even if a value is not written to on this execution, it should be tainted if it could have been written on the side of the branch that was not taken. However, we have found few cases that need the full complexity of this precise propagation, so we have currently implemented a simpler approach of tainting any value that is written by an instruction after the branch but before the post-dominator, with some special treatment for loops with a tainted loop condition. We discuss this trade-off more in Section 7.

Clause et al. apply taint propagation to all the conditional branches with a tainted condition (`eflags` value), but we apply it to only the culprit branches found by the diagnosis algorithm, which are very few (orders of magnitude less, as we show this in our results in Section 6). This allows our technique to provide more accurate tainting results.

4.3. Multi-level culprit implicit flows.

There can also be *multi-level culprit implicit flows* in the program: culprit implicit flows whose inputs are themselves the results of previous culprit implicit flows. Thus fixing all the culprit implicit flows visible in a single trace may still not be sufficient to resolve all the under-tainting in an execution. When needed our tool can repeat the combination of tracing, diagnosis and rule generation until we obtain a trace without under-tainting.

There are several possible approaches to detecting when to stop generating more propagation rules. For instance, if we have auxiliary information for the offline analysis phase specifying which output locations are under-tainted, we can generate rules until those locations are properly tainted. Alternatively, we can simply repeat the offline analysis process until it detects no remaining implicit flows.

5. Implementation

We implement our proposed technique using the BitBlaze platform [31]. BitBlaze includes tools for both dynamic and static analysis available in an open-source release [6]. We use our DTA++ implementation to extend and enhance the taint analysis performed by BitBlaze's dynamic analysis component TEMU and its Tracecap plugin.

```
/* check for culprit implicit flow */
HasImplicit(trace, size)
  if phi(trace[0:size])
    return true;
  else
    return false;
  endif

/* locating the first culprit branch  */
LocateImplicit(trace, 1, size_of(trace));
```

```
LocateImplicit(trace, left, right)
  integer middle, size;
  if left == right
    return right;
  else if left > right
    return -1;
  endif
  middle := (left + right) / 2;
  if HasImplicit(trace, middle) == true
    return LocateImplicit(trace, left, middle)
  else
    return LocateImplicit(trace, middle + 1, right)
  endif
```

Figure 6. Pseudo-code of the algorithms

Using TEMU's taint-enhanced whole-system emulation environment, Tracecap collects a trace that can include each instruction executed by a subject program, and information about its operands. To reduce the size of the generated trace we use Tracecap filtering features such as not starting the trace until the first instruction with a tainted operand. We also enhanced TEMU with additional interfaces for selectively tainting input values and verifying taint results of output data. For diagnosis we build on BitBlaze's existing support for path constraint generation (using the Vine toolkit) and constraint solving (with an interface to the off-the-shelf SMT decision procedure STP [19]). We implement the binary search algorithm of Section 4.1 and other glue code in Python. We obtain control flow graphs and post-dominator information by disassembling the program binary with IDA Pro [1], and then passing its output to a CFG library that is part of BitBlaze. The output of the rule generation phase is a text file containing DTA++ propagation rules. We then perform online taint propagation using these DTA++ rules to enhance dynamic taint analysis.

6. Word Processors Case Study

To evaluate our DTA++ approach, we apply our implementation to diagnose and fix culprit implicit flows in Windows word processors that exhibit under-tainting in format conversion. In this section we describe our experimental setup, evaluation metrics, and then our evaluation results.

6.1. Experimental Setup

For our evaluation we use 8 word processing applications that run on Microsoft Windows: Microsoft Word 2003 [23], WordPad [35], AbiWord [2], AngelWriter [3], Aurel RTF Editor [4], IntelliEdit [20], Crypt Edit [13], and VNU Editor. Each program can accept input in plain text format using the keyboard, and then save the text to disk in formats including RTF and HTML. For each application we first checked whether it had an under-tainting problem when converting plain text into either RTF or HTML, and if so, used our system to diagnose and fix the problem.

For each program with RTF output, we checked for under-tainting by running the program using TEMU's vanilla DTA and supplying a test input via the keyboard. For the experiments described in this section we use `"Taint it: {"` with all 11 characters tainted as the test input (other test inputs would give similar results). Then, we direct the program to save the text in RTF format and observed the tainting in the bytes written into the file. When the program converts the text to RTF, the program escapes the brace, yielding the result `"Taint it: \{"`. The desired tainting result is that all the characters except the backslash (\) are tainted. We identified cases as under-tainted if the brace character was not tainted in the output file, as illustrated in Figure 2. The experiments with HTML conversion are the same except that we use a less-than character < instead of a brace, which is escaped to `"&lt"`. (We did not observe under-tainting of any other bytes.)

Table 1 lists the combinations of program and target that showed under-tainting, which we evaluate with our tool. We then applied our DTA++ tool to diagnose and fix the under-tainting of the brace or less-than character respectively. None of these examples exhibited multi-level implicit flows, so only one execution of our diagnosis algorithm is required. Our execution platform is Windows XP SP3, running inside TEMU.

6.2. Evaluation Metrics

The most basic evaluation criterion for our technique is whether taint correctly propagates to the previously under-tainted output byte (e.g., `{`). In addition, we measure how many culprit implicit flows our system corrects, how much time it takes to do so, and how many total bytes are tainted (an indicator of over-tainting).

Number of culprit implicit flows diagnosed and fixed. We count the total number of culprit implicit flows diagnosed and fixed by our technique.

Program Description	# of Culprit Implicit Flows Detected & Fixed	Time for Diagnosis	Tainted Bytes (whole system)			
			Original	Optimal	DTA++	DYTAN*
WordPad, RTF	1	0.26s	90	131	139	25634
MS Word 2003, RTF	24	31m 5.26s	407	467	880	149485
AbiWord, HTML	1	0.63s	1062	1075	1289	89641
AngelWriter, HTML	3	14.29s	210	220	382	8503
Aurel Editor, RTF	1	0.76s	79	87	87	84425
VNU Editor, RTF	1	0.34s	101	120	121	18852
IntelliEdit, RTF	1	0.40s	127	132	132	12473
CryptEdit, RTF	1	0.23s	293	313	313	15509

Table 1. Program description and evaluation results

Program Description	# of Tainted Branches (# Unique Taint Branches)			
	Original	Optimal	DTA++	DYTAN*
WordPad, RTF	652 (64)	731 (84)	745 (87)	263292 (6248)
MS Word 2003, RTF	2620 (213)	2628 (244)	2685 (267)	417455 (15675)
AbiWord, HTML	4792 (356)	4825 (374)	5328 (446)	1024932 (11059)
AngelWriter, HTML	42 (11)	200 (49)	266 (96)	19808 (2269)
Aurel Editor, RTF	639 (63)	710 (77)	735 (88)	498904 (12134)
VNU Editor, RTF	921 (71)	1014 (89)	1039 (100)	74778 (4454)
IntelliEdit, RTF	1101 (98)	1190 (109)	1239 (123)	41898 (3114)
CryptEdit, RTF	744 (97)	822 (110)	823 (111)	57864 (3820)

Table 2. Number of tainted branches

Performance (time). Since symbolic execution and constraint solving are potentially expensive, we check that our technique does not add too much overhead. To assess this, we measure the time our system takes to locate the culprit implicit flows in each example, given an execution trace. (The overhead of additional propagation during future taint propagation runs would likely be too small to measure.)

Over-tainting evaluation. Some potential sources of over-tainting are outside the scope of this research, but it is important to be careful of over-tainting whenever we introduce additional taint propagation. So we check that our technique does not introduce excessive over-tainting as a side effect of fixing under-tainting. We measure the number of tainted bytes in the system memory after applying targeted taint propagation according to the rules generated by our offline analysis technique. For this purpose, we stop the analysis when the program writes its output to a file, and count the total number of tainted bytes in memory using Tracecap's state snapshot feature. We compare the number of tainted bytes between the unmodified execution and three types of propagation (as listed in Tables 1 and 2):

- Original: The starting point for comparison is the number of bytes that are tainted when executing the program using TEMU's vanilla DTA approach with no additional propagation. Of course this vanilla DTA has under-tainting.
- Optimal: For a best-case comparison of what results can be obtained by adding propagation, we inspect the execution trace and manually identify a single instruction that is responsible for under-tainting, and verify that adding taint to the values written by that instruction avoids under-tainting. Thus this measurement reflects the least possible additional tainting consistent with removing the under-tainting.
- DTA++: In this case we apply our DTA++ technique, using the targeted propagation rules generated using the techniques of Section 4.
- DYTAN*: To measure the value of targeting propagation, we compare to the results obtained with the simpler approach of performing propagation for every branch whose condition was tainted. This simulates using our infrastructure the results that would be obtained from a tool like DYTAN [10]. (We cannot compare directly with DYTAN because it does not support Microsoft Windows programs).

6.3. Results

Summary. We were able to automatically diagnose and ameliorate the under-tainting problems in all 8 programs using our implementation. To evaluate the accuracy and the efficiency of our technique, we use the metrics mentioned in the previous subsection. Table 1 presents the results. In summary, in most of the programs, our technique diagnoses a single culprit implicit flow, and just fixing the detected implicit flow solves the under-tainting problem in the output data. The two exceptions are the RTF conversion of Microsoft Word, and the HTML conversion of AngelWriter; in these cases our system finds multiple potential culprit branches though in fact only one is responsible for under-tainting. Our technique also diagnoses under-tainting problems efficiently. For most of the programs, the technique detects the implicit flows within one second. Microsoft Word is again an outlier in running time, largely because it has the largest number of implicit flows, and each execution trace contains many instructions. Counting the number of tainted bytes in the system memory shows that our targeted propagation reduces the unnecessary tainting dramatically compared to an approach that taints all control dependencies (presented as *DTA++* and *DYTAN** respectively in Table 1). Our technique taints 22 to 1445 times fewer bytes compared to an indiscriminate approach (DYTAN*). The amount of taint added by our automated approach is often quite close to the minimal additional taint we found by manual analysis (the "Optimal" column in Table 1).

We also count the number of tainted branches after fixing the culprit implicit flows. As shown in Table 2, DTA++ adds a few additional tainted branches by fixing the culprit implicit flows, indicating little over-tainting. However, indiscriminate propagation of DYTAN* shows up to 678 times more tainted branches, indicative of severe over-tainting.

Additionally, we have examined the execution traces to see exactly how the implicit flows affect taint propagation in our dynamic taint analysis. We select two subject programs to describe the under-tainting problems in detail.

WordPad RTF (a simple conditional branch). When converting a left brace (`{`) into the RTF format by prepending a backslash (\), WordPad first converts 1-byte characters into two-byte ones prefixed by 8 zero bits (i.e., converts them into Windows wide characters). During this conversion, the taint tagged on the original brace character does not propagate to the brace in the output buffer. Figure 7 shows exactly how the under-tainting problem develops in the WordPad program execution. In the course of RTF conversion, WordPad reads one character at a time, and checks if the current character is a left brace (`0x7b` in ASCII code) at EIP `0x4b44daad`. (The current character from the input is in the `al` register and tainted.) When the input value is equal to the value in the `dl` register (in this execution context, `0x7b`), it does another equality check against the null character (EIP: `0x4b44daab`). Since the input character is not null, the program execution gets through to `0x4b44dab5` where WordPad calculates an offset value needed to retrieve the two-byte value of the brace character. The offset value is from two untainted address values, and thus the resulting offset is not tainted. By using this untainted offset, WordPad retrieves the two-byte format of the left brace from a character table. Although TEMU propagates taint to values from an array access with a tainted index value, the taint doesn't propagate properly here because the index (offset) value is not tainted. Apparently, the calculation of the offset value is controlled by a tainted value (the tainted brace character from the input). However, the taint flows through the control dependency at `0x4b44dadd`, and thus, it doesn't propagate to the offset calculation and character conversion. Also, this program execution path is made possible only by the left brace character, so our diagnosis algorithm correctly detects it.

AbiWord HTML. AbiWord has an under-tainting problem when converting plain-text content to the HTML format. Unlike WordPad, the implicit flow causing the problem is not an explicit comparison against the character that AbiWord wants to convert. That is, in the WordPad example, the program explicitly compares the input character with the left brace (`0x7b`), but the comparison in AbiWord is more subtle. AbiWord converts a less-than sign into the `"&lt"` string in HTML. As in the execution trace in Figure 8, the program first compares the input character with the ampersand character (`&`) by subtracting `0x26`, the ASCII code for it (at `0x101eb133`). If it's not equal, the program compares the resulting value (`input - 0x26`) with `0x16`, which is to see whether the input value is a less-than character (<). In other words, it subtracts `0x3c` in total (`0x26 + 0x16`) from the input value instead of directly comparing the input value with `0x3c`. If the comparison passes at `0x101e13f`, the program pushes an address value into the stack. This address value is used to retrieve the converted string value (`"&lt"`) from a table later in the program execution. However, the address value is a constant value and so is not tainted by vanilla DTA.

7. Discussion

In this section we provide further discussion of some of the design choices made in our approach, and its limitations.

Symbolic memory index. In dynamic taint analysis we taint a value loaded from memory if the address used for the load is tainted; as mentioned in Section 3.1, this can be viewed as fixing a kind of implicit flow in memory accesses. (This treatment of loads is a configurable option in TEMU, but we enable it for our experiments.) At the source level, such memory loads often correspond to array indexing (where the address is derived from the array index), so we also refer to this as the case of a tainted memory index.

```
...
0x4b44daad: cmp  %dl,%al     ;%dl=0x7b (not tainted), %al=0x7b (tainted)
0x4b44daaf: jne  0x4b44daa6
0x4b44dab1: test %dl,%dl     ;%dl=0x7b (not tainted)
0x4b44dab3: je   0x4b44dacb
0x4b44dab5: sub  %edi,%ecx   ;%edi=0x4b4043a0 (not t.), %ecx=0x4b4043a2 (not t.)
0x4b44dab7: mov  0x4b4043ac(,%ecx,2),%ax  ;*0x4b4043b4=0x007b (not tainted)
0x4b44dabf: cmp  $0x7f,%ax   ;%ax=0x007b (not tainted)
...
```

Figure 7. Execution trace of an implicit flow in WordPad

```
...
0x101eb130: movsbl %bl,%eax        ;%bl=0x3c (tainted)
0x101eb133: sub    $0x26,%eax      ;%eax=0x0000003c (tainted)
0x101eb136: je     0x101eb223
0x101eb13c: sub    $0x16,%eax      ;%eax=0x00000016 (tainted)
0x101eb13f: je     0x101eb20e
0x101eb20e: push   $0x10281978
...
```

Figure 8. Execution trace of an implicit flow in AbiWord

However, this kind of taint propagation raises an additional question when we use taint to create symbolic values for path constraints: what is the correct constraint formula for the value read from the array? If we propagate taint through an array access with a tainted index, the value copied from the array also should be marked symbolic. The most precise approach would be to copy the entire contents of the array at the time of access into the constraint formula, but this has several practical problems. First, it is often difficult to determine the bounds of an array at the binary level. Second, large lookup-tables can make formulas too large and/or make them take too long to solve. Instead in our implementation, we make the variables for values read from symbolic inputs be free. In other words, we consider those tainted values as new input values without any constraints. This relaxation could potentially cause our tool to miss implicit flows, though this was not a problem in our experiments. We leave for future work the problem of how to account more accurately for such propagations in a way that is both automatic and scalable.

Symbolic indirect jump. It can also happen that taint propagates to an address used in an indirect jump. Somewhat like the case of symbolic indexes above, it is difficult to symbolically represent all the possible behaviors of such a jump, and they are also problematic in the construction of control-flow graphs. We did not encounter tainted indirect jumps in any of our examples, but the context in which they are most likely to occur is probably jump tables used to implement `switch` statements. For this limited case it would likely be feasible to implement special case recognition of common switch statement instruction patterns.

```
1 // table of n special characters:
2 table = {’{’, ’}’, ’\’, ...};
3 input = get_input();
4 for (i = 0; i < n; i ++) {
5     if (input == table[i])
6         break;
7 }
8 if (i < n)
9     output = "\" + table[i];
```

Figure 9. A negative implicit flow. Here the branch on line 5 affects the value of `i` because when the branch is taken, `i` is *not* modified.

Negative implicit flows. The most subtle type of implicit flows are those in which a tainted control flow affects later data because a value is *not* modified. Some authors reserve the term “implicit flow” for this narrower case; instead we distinguish them as *negative* implicit flows. Figure 9 shows an example of how such a flow can occur in text transformation; it is modeled after code we observed in CryptEdit. Negative implicit flows can be diagnosed by our technique just like other implicit flows, but they require a more sophisticated approach to generate propagation rules. Rather than just tainting locations that are written inside a branch, we must also analyze which values might have been written had the branch not been taken (e.g., the missing increments of `i` in Figure 9) and taint those locations as well. Our current implementation only handles such situations when, as in the example, they appear in relation to the exit condition of a loop. For this, we apply BitBlaze’s implementation of loop-extended symbolic execution [28], obtained from its authors. No other negative implicit flows caused under-tainting in our examples, but in the future we plan to extend our implementation to handle more negative implicit flows, at least in the most common cases without nested branches, arrays, or indirection.

Limitations of a dynamic approach. Unlike static analysis on either source code or a program binary, dynamic approaches can only diagnose under-tainting problems present in the instances of program execution that we observe. That is, the dynamic approaches cannot explore all the possible execution paths in the program, leaving unseen under-tainting problems. However, static approaches are often limited by the complexity of possible program states and path constraints.

We believe that a dynamic approach, based on test cases that exercise code that can suffer from under-tainting, is practical in this domain. This is because there are relatively few code locations responsible for under-tainting, and under-tainting is caused by the structure of the code so that complex prerequisite conditions are not required to trigger it. These intuitions are supported by our case studies so far. Another possibility would be to apply test-generation techniques to this problem: for instance, we could use symbolic execution or other techniques to automatically explore new paths and check them for under-tainting.

Forward vs. backward approaches. Our diagnosis technique is based on taint propagation information originated from input data, so, in some sense, it is a kind of forward slicing on the execution traces. However, since we can take the whole execution trace and know which part of the output data should be tainted, it would also be possible to reverse the process. When we find an untainted output byte that we expected to be tainted, we can attempt backward slicing from the sink to see how it tracks back to the corresponding tainted input data. However, backward slicing would not solve the problems of implicit flows that motivate this work. The same program constructs are generally referred to as control dependencies in the slicing literature, but they cause the same kinds of difficulties there that they do in tainting: ignoring control dependencies, as is often done in dynamic slicing, can yield slices that are too small, while including all control dependencies, as is common in static slicing, often yields slices that are too big. The best way to reconcile these tradeoffs is also the subject of research in slicing, yielding approaches such as “thin slicing” [32]. On a more practical level, a key advantage for us of forward rather than backward analysis is that forward taint propagation can be performed at the same time as the original forward execution, so our execution traces only must record tainted instructions. Backward slicing would require complete execution traces which would be much bigger: more than 1GB in some of our examples.

Malicious software components. Although we evaluated our technique only with benign applications, implicit flows in malicious software components can cause under-tainting. For instance, such under-tainting could affect the use of malware analysis platforms such as Panorama [37]. However these problems are harder to solve, since malware writers can deliberately generate and embed lots of conditional branches, invoking implicit flows of tainted input data. At a minimum, this means that by violating our assumption that implicit flows are relatively rare, an adversarial program author could make the techniques we present here be impractically slow. Other features of our current implementation could also be exploited to create false positive or false negative errors. To our knowledge these evasion techniques have not yet been seen in the wild, but we expect they would not be terribly difficult to implement if malware authors felt they were needed. At a minimum, this suggests that such systems should have a distinction between software components into those that are known benign, and those that might be malicious. Systems should then apply different and more conservative taint propagation policies to the potentially malicious components.

8. Conclusion

We have presented DTA++, an enhancement to dynamic taint analysis that additionally propagates taint along a targeted subset of control-flow dependencies. DTA++ allows dynamic taint analysis to avoid under-tainting when implicit flows occur in data transformations. By diagnosing culprit implicit flows and performing additional propagation only within information-preserving transformations, DTA++ resolves under-tainting without causing over-tainting. We have shown how our implementation of DTA++ applies to off-the-shelf Windows binaries. In a case study of 8 applications, DTA++ prevented under-tainting that would otherwise have given incorrect results, while introducing orders of magnitude less taint than when propagating taint for all implicit flows as in previous systems such as DYTAN [10].

Acknowledgments

The authors are grateful to Kevin Chen, Daniel Reynaud, and Aravind Iyer for their suggestions in improving the presentation of this paper.

This work was performed while Min Gyung Kang and Pongsin Poosankam were visiting student researchers at UC Berkeley. This material is based upon work partially supported by the National Science Foundation under Grants

No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under Grant No. 22178970-4170, by the Army Research Office under grant DAAD19-02-1-0389, and by the Office of Naval Research under MURI Grant No. N000140911081. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Air Force Office of Scientific Research, the Army Research Office, or the Office of Naval Research.

References

[1] The IDA Pro disassembler and debugger. http://www.hex-rays.com/idapro/.

[2] Abiword. `http://www.abisource.com/`.

[3] Angel Writer. `http://www.angelicsoftware.com/en/angel-writer.html`.

[4] Aurel RTF Editor. `http://sites.google.com/site/aurelwwiz/aurelsoft`.

[5] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–24, Trento, Italy, July 2010.

[6] BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.

[7] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Paris, France, July 2008.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, pages 321–336, San Diego, CA, USA, 2004.

[9] A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 334–339, Lisbon, Portugal, May 2007.

[10] J. Clause, W. Li, and R. Orso. Dytan: A generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, London, UK, July 2007.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Symposium on Operating Systems Principles (SOSP)*, pages 133–147, Brighton, United Kingdom, Oct. 2005.

[12] J. R. Crandall and Z. Su. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Computer and Communications Security (CCS)*, pages 235–248, Alexandria, VA, USA, Nov. 2005.

[13] Crypt Edit. `http://download.cnet.com/Crypt-Edit/3000-2079_4-10064884.html`.

[14] D. E. R. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, May 1975.

[15] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Annual Technical Conference*, pages 1–14, Santa Clara, CA, USA, June 2007.

[16] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley. Practical data confinement. Unpublished manuscript, `http://www.cs.berkeley.edu/~andreye/pdc_submission.pdf`, Nov. 2009.

[17] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley. Towards practical taint tracking. Technical Report UCB/EECS-2010-92, EECS Department, University of California, Berkeley, June 2010.

[18] J. S. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, Cambridge, UK, 1973.

[19] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, Berlin, Germany, July 2007.

[20] IntelliEdit. `http://www.flashpeak.com/inted/inted.htm`.

[21] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Progamming Language Design and Implementation (PLDI)*, pages 193–205, Tucson, AZ, USA, June 2008.

[22] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, Oakland, CA, USA, May 2007.

[23] Microsoft Word 2003. `http://msdn.microsoft.com/en-us/office/aa905483.aspx`.

[24] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[25] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2007.

[26] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2005.

[27] Rich text format (RTF) specification, version 1.6. `http://msdn.microsoft.com/en-us/library/aa140280(office.10).aspx`.

[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)*, Chicago, IL, July 2009.

[29] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2010.

[30] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *European Conference on Computer Systems (EuroSys)*, pages 61–74, Nuremberg, Germany, Apr. 2009.

[31] 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 (keynote invited paper). In *International Conference on Information Systems Security (ICISS)*, Hyderabad, India, Dec. 2008.

[32] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Progamming Language Design and Implementation (PLDI)*, pages 112–122, San Diego, CA, USA, June 2007.

[33] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 89–108, Lucerne, Switzerland, July 2007.

[34] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, Boston, MA, USA, Oct. 2004.

[35] Wordpad. `http://windows.microsoft.com/en-US/windows-vista/Using-WordPad`.

[36] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2008.

[37] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Computer and Communication Security (CCS)*, Alexandria, VA, USA, Oct. 2007.