Dytan: A Generic Dynamic Taint Analysis Framework

James Clause, Wanchun Li, and Alessandro Orso
College of Computing
Georgia Institute of Technology
{clause|wli7|orso}@cc.gatech.edu

ABSTRACT

Dynamic taint analysis is gaining momentum. Techniques based on dynamic tainting have been successfully used in the context of application security, and now their use is also being explored in different areas, such as program understanding, software testing, and debugging. Unfortunately, most existing approaches for dynamic tainting are defined in an ad-hoc manner, which makes it difficult to extend them, experiment with them, and adapt them to new contexts. Moreover, most existing approaches are focused on data-flow based tainting only and do not consider tainting due to control flow, which limits their applicability outside the security domain. To address these limitations and foster experimentation with dynamic tainting techniques, we defined and developed a general framework for dynamic tainting that (1) is highly flexible and customizable, (2) allows for performing both data-flow and control-flow based tainting conservatively, and (3) does not rely on any customized run-time system. We also present DYTAN, an implementation of our framework that works on x86 executables, and a set of preliminary studies that show how DYTAN can be used to implement different tainting-based approaches with limited effort. In the studies, we also show that DYTAN can be used on real software, by using FIREFOX as one of our subjects, and illustrate how the specific characteristics of the tainting approach used can affect efficiency and accuracy of the taint analysis, which further justifies the use of our framework to experiment with different variants of an approach.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging;

General Terms: Experimentation, Security

Keywords: Dynamic tainting, information flow, general framework

1. INTRODUCTION

Dynamic taint analysis (also known as dynamic information flow analysis) consists, intuitively, in marking and tracking certain data in a program at run-time. This type of dynamic analysis is becoming increasingly popular. In the context of application security, dynamic-tainting approaches have been successfully used to prevent a wide range of attacks, including buffer overruns (e.g., [8, 17]), format string attacks (e.g., [17, 21]), SQL and command injections (e.g., [7, 19]), and cross-site scripting (e.g., [18]). More recently, researchers have started to investigate the use of tainting-based approaches in domains other than security, such as program understanding, software testing, and debugging (e.g., [11, 13]).

Unfortunately, most existing techniques and tools for dynamic taint analysis are defined in an ad-hoc manner, to target a specific problem or a small class of problems. It would be difficult to extend or adapt such techniques and tools so that they can be used in other contexts. In particular, most existing approaches are focused on data-flow based tainting only, and do not consider tainting due to the control flow within an application, which limits their general applicability. Also, most existing techniques support either a single taint marking or a small, fixed number of markings, which is problematic in applications such as debugging. Finally, almost no existing technique handles the propagation of taint markings in a truly conservative way, which may be appropriate for the specific applications considered, but is problematic in general. Because developing support for dynamic taint analysis is not only time consuming, but also fairly complex, this lack of flexibility and generality of existing tools and techniques is especially limiting for this type of dynamic analysis.

To address these limitations and foster experimentation with dynamic tainting techniques, in this paper we present a framework for dynamic taint analysis. We designed the framework to be general and flexible, so that it allows for implementing different kinds of techniques based on dynamic taint analysis with little effort. Users can leverage the framework to quickly develop prototypes for their techniques, experiment with them, and investigate trade-offs of different alternatives. For a simple example, the framework could be used to investigate the cost effectiveness of considering different types of taint propagation for an application.

Our framework has several advantages over existing approaches. *First*, it is highly flexible and customizable. It allows for easily specifying which program data should be tainted and how, how taint markings should be propagated at run-time, and where and how taint markings should be checked. *Second*, it allows for performing data-flow and both data-flow and control-flow based tainting. *Third*, from a more practical standpoint, it works on binaries, does not need access to source code, and does not rely on any customized hardware or operating system, which makes it broadly applicable.

We also present DYTAN, an implementation of our framework that works on x86 binaries, and a set of preliminary studies performed using DYTAN. In the first set of studies, we report on our experience in using DYTAN to implement two tainting-based approaches presented in the literature. Although preliminary, our experience shows that we were able to implement these approaches completely and with little effort. The second set of studies illustrates how the specific characteristics of a tainting approach can affect efficiency and accuracy of the taint analysis. In particular, we investigate how ignoring control-flow related propagation and overlooking some data-flow aspects can lead to unsafety. These results further justify the usefulness of experimenting with different variations of dynamic taint analysis and assessing their tradeoffs, which can be done with limited effort using our framework. The second set of studies also shows the practical applicability of DYTAN, by successfully running it on the FIREFOX web browser.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

The contributions of this paper are:

- The definition of a generic framework for dynamic taint analysis that (1) suitably handles information flow due to data and control flow within a program and (2) allows for customizing the analysis along several dimensions.
- A tool, DYTAN, that implements our framework for x86 executables, works at the application level, and does not require any special support from the runtime system.
- A set of studies that provide initial evidence of the generality of our framework, its applicability, and the potential usefulness of experimenting with different variations of dynamic taint analysis.

The rest of the paper is organized as follows. We provide background information and motivation for the work in Section 2. Section 3 discusses related work. Section 4 presents our approach and the tool that implements our approach. We discuss our empirical evaluation of the approach in Section 5. Finally, we conclude and discuss future-work directions in Section 6.

2. BACKGROUND AND MOTIVATION

In this section, we present background information on dynamic tainting and a set of examples that we use throughout the paper. Intuitively, dynamic tainting tracks the information flow within a program by (1) associating one or more markings with some data values in the program and (2) propagating these markings as data values flow through the program during execution. Consider, for instance, the simple example in Figure 1. Imagine that we tainted the variables `a` at line 2 and `b` at line 3 with taint markings t_a and t_b, respectively. In such a case, we would expect, at the end of the execution, that the taint markings associated with variables `x`, `y`, and `z` would consist of sets $\{t_a\}$, $\{t_b\}$, and $\{t_a, t_b\}$, respectively. Taint marking t_a, initially associated with `a`, would be associated with `w` because `a`'s value is used to compute `w`. Analogously, marking t_a would be associated with `y` because the value of `w`, which is now tainted with t_a, is used to compute `y`. The propagation of taint markings for the remaining variables is analogous.

```
1  int a, b, w, x, y, z;
2  a = 11;
3  b = 5;
4  w = a * 2;
5  x = b + 1;
6  y = w + 1;
7  z = x + y;
```

Figure 1: Example code that contains explicit information flow.

In this example, taint propagation occurs because of *explicit information flow*, that is, direct involvement of a tainted variable in the computation of another variable's value. Explicit information flow can also be described as propagation that occurs due to data flow or data dependences[1] in the code (e.g., there is a data-flow chain between `y` and `a`).

A less intuitive cause of propagation is *implicit information flow*, which refers to situations in which a tainted data value affects the value of another variable indirectly. Whereas explicit information flow is related to data dependencies, implicit information flow is typically due to control dependences[2] in the code. For an example, consider the code in Figure 2(a) and assume that we tainted the value of parameter `a` at line 1 with taint marking t_a. Although `a`'s value is not involved in the computation of `x`, it nevertheless affects `x`'s value through a control dependence: the outcome of the predicate at line 3 decides whether line 4 or line 7 will be executed. Therefore, the value of `x` at the end of the execution should be tainted with marking t_a. Conversely, variable y would not be tainted because its value does not depend on a's value in any way.

```
 1  void foo(int a) {
 2    int x, y;
 3    if (a > 10) {
 4      x = 1;
 5    }
 6    else {
 7      x = 2;
 8    }
 9    y = 10;
10    print(x);
11    print(y);
12  }
```

(a)

```
 1  void foo(int a) {
 2    int x, y;
 3    x = 2;
 4    if (a > 10) {
 5      x = 1;
 6    }
 7    y = 10;
 8    print(x);
 9    print(y);
10  }
```

(b)

Figure 2: Example code that contains implicit information flow.

Note that there are even subtler cases of implicit information flow. For example, consider the code in Figure 2(b), which is semantically equivalent to the code in Figure 2(a) but does not contain an `else` branch for the `if` statement at line 4. Assume that we invoke procedure `foo` as `foo(2)`. In this case, the predicate (now at line 4) would still affect the value of `x`. However, this information flow would be difficult to identify by simply observing the execution because `x` is defined before the predicate is computed and is not redefined afterward.

As we discuss in detail in Section 4, our approach takes into account both explicit and implicit information flow in a conservative way, so allowing the user to perform an accurate and safe dynamic taint analysis. Before describing our approach, we present some applications of dynamic taint analysis that could benefit from our framework and discuss related work.

3. RELATED WORK

There is a good deal of related work for our approach. We discuss the most closely-related approaches and provide a quick overview of additional related work. Note that, with the exception of the techniques discussed in Section 3.1, most of the techniques discussed in the rest of this section are not alternative approaches to ours, but rather possible applications of our framework.

3.1 General Approaches to Dynamic Tainting

Due to the increased popularity of dynamic tainting, there have been a few recent attempts at providing a generalized tainting infrastructure [10, 25]. These techniques are meant to be used in the security field and, thus, have limitations that prevent their use in more general contexts. Lam and Chiueh [10] propose an approach that instruments the code to perform taint marking and propagation. Their approach has two main drawbacks compared to ours. First, it requires the code to be recompiled, which is especially problematic when third-party and system libraries are involved (as it is typically the case with real software). Second, their approach lacks support for control-flow based tainting. While it is true that most security applications of dynamic tainting only require data-flow based tainting, we believe that a truly general framework should support both types of propagation. As we show in Section 5, control-flow propagation can have dramatic effects on the results of dynamic taint analysis. The framework proposed by Xu and colleagues [25] shares the same two drawbacks of Lam and Chiueh's approach and has the additional limitation of not supporting multiple taint marks.

[1] A statement s_2 is data dependent on a statement s_1 if (1) s_2 uses a variable v that is defined in s_1 and (2) there is a def-clear path with respect to v between s_1 and s_2.

[2] Informally, a statement s_2 is control dependent on a statement s_1 if (1) s_1 contains a predicate, and (2) depending on the outcome of s_1, s_2 may not to be executed.

3.2 Attack Detection and Prevention

Dynamic tainting has been used extensively to detect attacks targeting software vulnerabilities. The most studied type of software exploit is *overwrite attacks*, a class of attacks where sensitive program data is overwritten by an attacker. The data overwritten typically consists of return addresses, function pointers, or format strings. By suitably overwriting this data, attackers are able to hijack a program and execute arbitrary code. The two most common types of overwrite attacks are buffer overflows and format string attacks.

Newsome and Song [17] present one of the first dynamic-taint based approaches for preventing overwrite attacks. Their approach taints any data read from a network socket. The tainted data is then propagated as the program executes. Finally, the approach enforces the security of a program by checking that tainted data is not used as the target of a jump (including function return addresses), a format string, or a system-call argument. Several other techniques for detecting overwrite attacks were developed at a similar or later time. In particular, Suh and colleagues [22] and Kong and colleagues [8] propose hardware-based approaches. More recent work has focused on reducing the overhead of earlier approaches. LIFT [21], in particular, attempts to reduce the overhead of propagating taint information by using a direct mapping between memory and taint labels. It also proposes several optimizations that eliminate unnecessary taint propagation operations.

Dynamic tainting has also been used to prevent *SQL injection attacks*, in which attackers submit maliciously-crafted strings to a web application to access its underlying database. Most dynamic-taint based approaches against SQL injection also operate by tainting and tracking unsafe data (i.e., input from the user). Then, before a query string is sent to the database, it is checked to ensure that no tainted data was used to create the string or specific parts of it. Nguyen-Tuong and colleagues [18] propose an instance of this approach for PHP-based web applications, whereas Haldar, Chandra, and Franz [6], Pietraszek and Berghe [19], and Halfond and Orso [7] target applications written in Java.

3.3 Information-flow Policies Enforcement

Dynamic tainting has also been successfully used in the context of information-flow security to enforce information-flow policies. Such policies define limits on how information is used within a system. An example of information-flow security policy in the physical world is a military system where classified information is forbidden to be transferred to individuals without the appropriate clearance level. Dynamic tainting is an ideal technique for enforcing information-flow policies in a software system; different taint markings can be used to label sensitive information and then the analysis can check whether marked data reaches parts of the system that it is not supposed to reach according to the policies in place.

Also in this case, there are several variations of this general approach. The RIFLE system [23] provides architecture-based support for information security by tracking explicit and implicit information flows. Chow and colleagues [3] present TAINTBOCHS, a simulator that can track tainted data through an entire system, including hardware, operating system, and applications. Using their system, they investigate the data lifetime of sensitive information in several commonly-used applications. Finally, McCamant and Ernst [14] present a technique that produces an upper bound of the amount of information leaked by a program at runtime.

3.4 Software Testing and Debugging

Outside of the security field, researchers have started to investigate the use of dynamic taint analysis in the areas of software testing and debugging. The `COMET` system [11] uses dynamic tainting combined with heuristic search to increase statement coverage for C programs. Dynamic tainting is used to construct a taint graph that represents how function inputs relate to variables used in conditions. With this information, test input generation can focus on modifying the outcome of specific conditions by narrowing the search space to include only influential inputs.

Masri and colleagues [13] propose an approach for identifying and debugging insecure information flows based on dynamic tainting and dynamic slicing. The dynamic tainting portion of the approach is used to detect illegal information flows under a specific security policy. Once an illegal flow has been detected, the dynamic slicing portion of the approach is used to extract the relevant portion of the execution (i.e., the set of statements that propagated information along the illegal flow). This set of statements can then be used to reduce the amount of code that needs to be examined when debugging the insecure flow.

3.5 Additional Related Work

Two additional areas of related work are static information-flow analysis and dynamic slicing. *Static information-flow analysis* (e.g., [16, 20]) is the static counterpart of dynamic taint analysis. As it is typically the case, static information-flow analysis has the advantage of computing conservative estimates of the information-flow within a program, whereas dynamic tainting can only identify flows that actually occur in one of the observed executions. On the flip side, static information-flow analysis can produce many spurious results in the presence of constructs such as loops and aliases, due to imprecision. Which approach is preferable depends on the specific application considered.

Another related approach is *dynamic slicing* (e.g., [1, 9]), which computes a conservative estimate of all statements in a program that are either affected by or affecting the value of a variable at a specific program point and for a given execution. Dynamic slicing and dynamic taint analysis are similar in nature, but compute slightly different kinds of information. The former identifies the subset of the statements in a program that affect (or are affected by) one or more data values. The latter focuses on computing which subset of the data in the program is affected by a given set of data.

4. OUR APPROACH

In this section, we describe our generic approach for dynamic taint analysis. More precisely, we provide (1) a general description of our framework, (2) details on the instantiation of the general approach for x86 binaries, and (3) a description of the tool that implements the framework.

4.1 General Framework

There are three dimensions that characterize dynamic taint analysis: taint sources, propagation policy, and taint sinks. Because different dynamic taint analyses can be expressed by defining the analysis along these three dimensions, we defined our general framework in terms of these dimensions.

Taint Sources. *Taint sources* are a description of program data (memory locations) that should be initialized with taint markings. Memory locations can be of different types, including variable names, function-return values, and data read from and I/O stream such as a file or a network connection. Our framework allows for specifying memory locations as follows:

- **Variables and memory offsets.** Users can indicate that a memory area should be tainted by specifying the corresponding variable name and scope (i.e., global or local to a procedure). For example, a user could specify variable `v` in procedure `foo` as a taint source. This would cause the area of memory that corresponds to v to be tainted during execution. Users can also directly specify a specific area of memory in terms of its offset from the base address of the program.

Data returned from a specific function. Users can indicate that a function's return value must be tainted by simply specifying the name of the function. For example, imagine an application that reads data from a database using a function "submitQuery"; a user would be able to taint all data originating from the database by specifying "submitQuery" as a source.

Data from a type of I/O stream. To indicate that data read from a particular type of stream must be tainted, users only need to specify the stream type. Currently, our framework handles three types of I/O stream: network, filesystem, and keyboard. For example, users could specify that they want any data coming from the keyboard to be tainted.

Data from a specific I/O stream. Users can also specify that data from a specific I/O stream must be tainted. In this case, only the network and the filesystem, for which it makes sense to distinguish between different streams, are supported. To indicate the specific stream of interest, users can specify its type, network or filesystem, and a unique identifier (absolute path for files, IP address or IP name and port for network streams).

In addition to specifying the specific program data to be tainted, a taint source must also indicate how taint markings should be associated with the identified memory areas. One possibility is to use a single taint mark. Many existing dynamic taint analyses, including most of the techniques discussed in Section 3, can be implemented using a single taint marking. Other applications may need to discriminate between data read from different sources. An obvious example are techniques for information-flow policy checking and enforcement, which may need to distinguish, for instance, between data coming from network hosts with different levels of trust. To support these applications, our framework allows for specifying different taint marking for different sources.

Additionally, for taint sources associated with I/O streams, our framework lets users associate different taint markings to different data read from a given stream. In this case, users would have to specify that they want to use "fresh markings" for that stream and specify the amount of data to be tainted with a single marking. Our framework would then assign a newly-created taint marking to each block of data of the given size read from the specified stream and keep track of the mapping between the block number and the associated taint marking. Note that, because of the way we store taint markings, we can allocate virtually any number of taint markings (obviously, at the cost of space overhead).

Propagation Policies.

A *propagation policy* describes how taint markings should be propagated during execution. Taint propagation can be expressed in general terms as follows: given a statement s, the taint markings for the data produced by s (*produced data*) are some function (*mapping function*) of the taint markings associated with the data that affects the outcome of s (*affecting data*). The produced data can be identified unambiguously—it is the set of data, stored in registers or memory, whose value is modified as a consequence of executing s. Conversely, there are different ways in which to identify affecting data and define the mapping function. Because our goal is to provide enough flexibility in our framework, we let users specify both aspects of a propagation policy.

Identifying affecting data. In our framework, users can specify one of two ways of identifying affecting data: data-flow based and data- and control-flow based. The former accounts only for explicit propagation of taint markings, which occurs through direct or transitive value assignments (see example in Figure 1). The latter also accounts for implicit propagation, which occurs due to control-flow dependences (see examples in Figure 2). Based on the application they are targeting, users can specify which of these two schemes to use. Note that data- and control-flow based propagation induces a higher runtime overhead than propagation based on data-flow only, as shown in Section 5.1.2. Therefore, if a technique can be implemented without considering implicit propagation, it can be more efficient. Using our framework, a user could experiment with both possibilities and assess the trade-offs of the different solutions.

Defining a mapping function. Typically, the set of affecting data contains several data items with multiple taint markings. In these cases, the default behavior of our framework is to taint the produced data with a set containing the union of all such markings. However, there are many ways in which taint markings can be propagated depending on the specific application of dynamic tainting considered. For example, one technique may need to keep a set of distinct taint markings for each data item, another technique may merge markings based on some predefined subsume hierarchy among them, and yet another technique may generate new markings for the produced data based on the specific set of markings of the affecting data. Currently, we allow users to define these custom mapping functions by redefining the procedure in the framework that combines the markings. Details about the specific way in which this is accomplished in the current implementation of our framework are provided in Section 4.3.

Taint Sinks.

At a high level, a *taint sink* is a location in the code where users want to perform some check on the taint markings of one or more memory locations. Taint sinks are characterized by four aspects: (1) an ID, (2) a memory location, (3) a code location, and (4) one or more checking operations to be performed at that code location and using the taint marking(s) associated with that memory location. The ID is an integer value assigned by the user to a sink or group of sinks. We explain the purpose of the IDs below, when discussing checking operations. Our framework provides two main ways to specify a sink's memory location and code location.

In the *first way*, memory and code locations are specified independently. In this case, like for taint sources, users can specify variables and memory locations using a variable's name and scope, indicating a memory area directly, specifying a procedure name and the index of the parameter (for formal parameters), or specifying the name of the function (for return values). The code location can also be specified in different ways. Under some conditions, users can specify the code location in terms of its position in the source code. (Because we want our framework to be able to operate at the binary level, this option is not always viable; it requires the binary code to contain debugging symbols.) Alternatively, users can specify the location in the binary code, in terms of its offset from the base address of the program. To simplify the use of our framework, we also give users the possibility of indicating the entry or the exit points of a procedure, specified by name, as the location of interest.

The *second way* accounts for scenarios where users wants to analyze the taint information before *each* instruction of a given type (e.g., a system call or a jump instruction). In this case, users can simply specify the instruction of interest, and their checking operation gets invoked right before any instruction of that type is executed. The taint markings associated with the data read by the instruction are passed as a parameter to the checking operation, and the same sink ID gets associated to all instructions.

A checking operation for a taint sink is provided as a user-specified function. The function must accept as input three parameters: an

```
<foo+00>:    <function preamble>
<foo+06>:    cmpl   $0xa,8(%ebp)
<foo+10>:    jle    0x15 <foo+21>
<foo+12>:    movl   $0x1,-16(%ebp)
<foo+19>:    jmp    0x1c <foo+28>
<foo+21>:    movl   $0x2,-16(%ebp)
<foo+28>:    movl   $0xa,-12(%ebp)
<foo+35>:    mov    -16(%ebp),%eax
<foo+38>:    mov    %eax,(%esp)
<foo+41>:    call   0x3b <dyld_stub_print>
<foo+46>:    mov    -12(%ebp),%eax
<foo+49>:    mov    %eax,(%esp)
<foo+52>:    call   0x3b <dyld_stub_print>
<foo+57>:    leave
<foo+58>:    ret
```

Figure 3: Assembly code for the code in Figure 2(a).

ID, a memory address, and a special data structure. At runtime, the memory address will point to the address associated with the sink, and the data structure will contain the taint information (i.e., the set of taint markings) associated with that address at the code location specified. If the checking operation just needs to know the taint information, then it would not use the second parameter. In some cases, however, a checking function may need to access the actual variable or memory location (e.g., to check the taint markings associated with different parts of the variable, for variables longer than one byte). To this end, our framework provides an API that can be used to retrieve taint information. The ID allows, for instance, for sharing a checking function among different sinks and having a slightly different behavior for different sinks.

For a simple example of how taint sources, propagation policies, and taint sinks work together, consider again the code in Figure 2(a). Assume that a user of our framework is interested in checking whether the value of parameter `a` affects in any way the value of `y` printed at the end of function `foo`. In this case, the user would specify the first parameter of `foo` as the taint source and indicate that it should be marked with a given taint marking t_a. The user would then select the default data- and control-flow based propagation policy. The user would finally specify a taint sink consisting of variable `y` at line 11, and whose associated procedure takes as input the taint information for `y` at line 11 and checks whether it includes taint marking t_a (or simply logs that taint information for a subsequent check).

4.2 Instantiation for x86 Binaries

Although the abstract definition of dynamic taint analysis is relatively straightforward, defining the details of the approach so that it is sound and accurate is considerably more complex. In this section, we define the instantiation of our generic framework for dynamic taint analysis for binaries running on the x86 architecture. In the discussion, we focus mainly on the details of data- and control-flow based propagation, which represent the core of dynamic taint analysis and are the most challenging parts of the approach.

Before discussing the details of the approach, we present a brief overview and example of the x86 assembly language. Figure 3 shows the assembly code for the code in Figure 2(a). Individual x86 instructions consist of a mnemonic command name followed by a variable number of operands. Operands can be registers (e.g., `%eax`, `%esp`), literal values (e.g., `$0xa`, `$0x2`), or a memory location (e.g., `-12(%ebp)`, `<foo+21>`). For example, the instruction at address `<foo+21>`, which corresponds to line 7 in the source code of Figure 2(a), copies the literal value 2 into the address at offset `-16` relative to `%ebp`, which corresponds to variable `x`.

For each assembly statement, the mnemonic determines the *source operands* (i.e., operands whose value is used by the statement to perform its computation) and *destination operands* (i.e., operands whose value is modified during the computation). For example, the assembly instruction `mov` takes two operands and copies the value of the first operand (source) into the second operand (destination). For another example, the `add` instruction computes the sum of its two operands, treating both as sources, and stores the result in the second operand, which is, thus, also a destination.

4.2.1 *Maintaining Taint Markings*

Performing dynamic taint analysis involves storing taint information for data items within the program being analyzed. In our approach, taint markings are stored in bit vectors, which each bit representing a different taint marking. Using bit vectors is a standard way to limit the cost of combining taint markings in the (predominant) case where the combination is defined as the union of the markings. We associate one bit vector to each x86 register and to each tainted memory location.

In our current implementation, the granularity that we consider for tainting is a byte. In our initial investigation, we did not find tainting at finer granularity, such as the bit level, to be cost effective. Nevertheless, it would be fairly straightforward to modify the approach so that it operates at a finer granularity, if further experiences show that it is needed.

4.2.2 *Data-flow Based Taint Propagation*

Our high-level approach to data-flow based propagation consists of two steps: given an assembly instruction, (1) identify the source and destination operands based on the instruction mnemonic and (2) combine the taint markings associated with the source operands and associate them with the destination operands. (As we explained above, the way in which taint markings are combined may vary based on the application, and the default behavior is to union them.) When this general approach is applied to x86 instructions, there are several problems (and opportunities) that must be considered for the dynamic taint analysis to be conservative and accurate. In the rest of this section, we discuss the most important of these aspects.

Mapping between Sources and Destinations. For some x86 instructions, different subsets of the source operands affect different subsets of the destination operands. For example, the `push` instruction, which stores the value of a register on the stack, has the register to be pushed and the stack pointer as source operands, and the memory location indicated by the stack pointer and the stack pointer itself as destination operands. Because the stack pointer is decremented by a constant value, it is not affected by the source register, whereas the memory location where the register value is stored is affected by both source operands. In such cases, propagating taint marks from all sources to all destinations would result in more markings than necessary being propagated and would introduce unnecessary imprecision in the analysis. Our approach considers the semantics of the different instructions, suitably identifies which source operands affect which destination operands, and propagates taint marks accordingly.

Address Generators. This issue is related to the use of memory locations as operands. The x86 architecture supports several different addressing modes that can be used to access memory. In general, addressing modes are either direct, where the memory location is specified using a constant, or computed using some combination of register values and constants. For example, in instruction `jle 0x15` (position `<foo+10>` in Figure 3), `0x15` directly specifies the target of the jump instruction as an offset of the current location. For a different example, in instruction `movl $0xa,-12(%ebp)` (position `<foo+28>`), the target of the operation is expressed as the value of register %ebp minus 12. Registers used to compute memory locations are commonly referred to as *address generators*.

If a memory location is specified using a constant, considering only the taint markings associated with that memory location is safe. However, if a memory location is specified using a combination of registers and constants, considering the tainting of the location alone is not enough; to be conservative, the taint markings associated with the address generators should also be considered. Most dynamic tainting approaches for preventing buffer overruns do not consider taint markings associated with address generators because they can be safely discarded for that specific application. However, these approaches are not safe in general. Because our goal is to provide a general, sound framework for taint propagation, we account for these possible sources of tainting in our approach.

Implicit Operands. In x86 code, not all operands can be identified by simply looking at the code. More precisely, an x86 statement can have *explicit operands*, which are present in the statement, and *implicit operands*, which are read or modified by the statement without being explicitly present. The set of implicit operands accessed by an instruction depends on the semantics of the instruction. Therefore, modeling such semantics is the only way to correctly account for implicit operands. For example, consider instruction push %eax, which has the following semantics: first subtract 4 from register %esp (stack pointer), then store the content of register %eax into the memory location stored in %esp. This instruction involves (1) an explicit read of %eax, (2) an implicit write to %esp, and (3) an implicit read of the memory location whose address is stored in %esp. A taint analysis that does not consider the two implicit operands (%esp and the memory location) would not propagate taint markings associated with register %eax to the memory location on the stack and would be unsafe.

This issue is especially relevant when the %eflags register is an implicit operand. For example, consider instruction add %eax, %ebx. In addition to calculating the sum of its explicit operands (i.e., %eax and %ebx), add also implicitly defines several bits in the %eflags register. In general, the %eflags register is used to keep some "state" information about the computation. In particular, after the execution of most arithmetic instructions, the %eflags register indicates the parity of the result, whether an overflow occurred, the sign of the result, and whether the result was zero. Because the value of %eflags is used by conditional-jump instructions, correctly propagating taint markings to the %eflags register is a prerequisite for correct control-flow propagation of taint markings, as explained in Section 4.2.3.

Sub-registers. For backward compatibility, the newer 32-bit architectures map the lower half of their 32 bit general-purpose registers to 16 bit registers available in older (16-bit) architectures. For example, the lower 16 bits of the 32 bit register %eax can be directly accessed as register %ax. To further complicate the picture, each of the lower two bytes of some 32 bit registers can also be accessed directly (e.g., bits 0–7 of register %eax can be accessed as register %al, and bits 8–15 can be accessed as register %ah). Although these direct addressing modes were implements to support legacy applications, they are also used in new applications to handle smaller data types and perform string-processing operations. A dynamic tainting approach that does not account for the presence of directly addressable sub-registers would be unsafe; it would fail to recognize (and suitably handle) the fact that, for instance, registers %ah and %eax are overlapping. When retrieving taint markings for a 32-bit registers, our approach also considers the taint markings for all sub-registers. Analogously, when retrieving taint markings for a 16-bit sub-register, it also considers the taint markings for the 32-bit register that contains the sub-register.

Constant Functions. Constant functions are sequences of instructions that always produce the same result regardless of their input values. An example is the x86 idiom for clearing a register (e.g., %eax): xor %eax, %eax. After executing this instruction, the %eax register always contains the value 0. There are several other instances of single-instruction constant functions, such as sub %eax, %eax and mov %eax, %eax. For these instructions, the safe data-flow propagation policy of assigning to the destination operands the combination of the taint markings associated with the source operands is safe, but can introduce considerable imprecision. To reduce this imprecision, we carefully studied the x86 instruction set and related manuals [5] to identify constant functions and encode their semantics into our framework.

Compound Branch Instructions. Compound branch instructions are single instructions that include control flow. An example is cmov <src> <dest>, which copies the value of its <src> operand into its <dest> operand only if a specified bit of the %eflags register is set (or unset, depending on the specific variant of cmov used). An imprecise propagation would always propagate taint markings from <src> to <dest>, whereas a more precise analysis can check the relevant %eflags's bit and propagate taint markings only when appropriate.

4.2.3 Control-flow Based Taint Propagation

As discussed in Section 2, taint markings can propagate explicitly, due to data flow, or implicitly, due to control flow. We now present a general approach for control-flow based taint propagation and discuss how we instantiated the approach for x86 binaries.

Background. First, we concisely introduce a few background concepts that we need to present the approach. A *Control Flow Graph (CFG)* is a directed graph whose nodes represent statements, whose edges represent possible flow of control between statements, and that contains two special nodes $entry$ and $exit$ with no predecessors and no successors, respectively. Given two nodes m and n in a CFG, n *postdominates* m (n *pdom* m or $pdom(m) = n$) iff all directed paths from m to $exit$ contain n. Given two nodes m and n in a CFG, $n\ immediately\ postdominates\ m$ ($n\ ipdom\ m$ or $ipdom(m) = n$) iff n pdom m and there is no node o such that n pdom o and o pdom m. A *postdominator (pdom) tree* for a CFG is a rooted tree such that (1) it has the same set of nodes as the CFG, (2) its root is the $exit$ node, and (3) each node immediately postdominates its direct descendents in the tree. Finally, given two nodes m and n in a CFG, n is *control dependent* on m iff (1) There is a path P from m to n with any node o in P (excluding m and n) postdominated by n, and (2) m is not postdominated by n.

General approach. The general approach we use is similar to other approaches presented in the literature (e.g., [13, 23]) and is based on the concept of control dependence. As we discussed in Section 2, indirect propagation occurs due to control dependences between statements. When a conditional branching statement br decides whether a statement st may be executed, the values that affect br's outcome may affect the value of the data modified by st. Therefore, to be conservative, the taint markings associated with br's source operands must be combined and associated with st's destination operands. To achieve this result, our approach keeps track at runtime of relevant taint markings by leveraging statically-computed postdominance information, as follows.

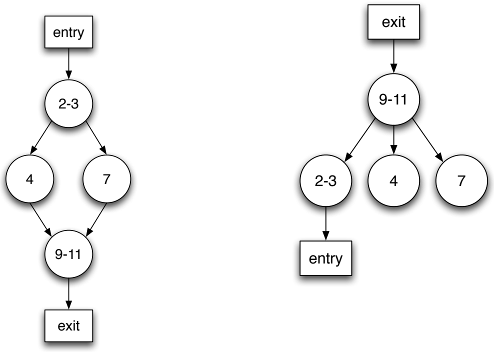

Figure 4: CFG and pdom tree for the code in Figure 2(a).

- When the execution reaches a conditional branching statement br, our approach (1) computes a set $taint$ that contains the combination of the taint markings for br's source operands and (2) add to a set S a pair $< br, taint >$.
- When the execution reaches $ipdom(br)$, where br is a conditional branching statements, it removes from S all pairs $< x, y >$ such that x is equal to br.
- When a statement st is executed and S is not empty, it adds, for each pair $< x, y >$ in S, the taint markings in y to the set of taint markings to be combined and associated to st's destination operands.

Because (by definition) all of the statements control dependent on a conditional branching statement br are on paths that start at br and end with br's immediate postdominator, the approach described above is guaranteed to conservatively propagate taint markings according to the control dependences in the program. To provide an illustrative example, we use again the code in Figures 2(a) and 3, whose CFG and postdominator tree are shown in Figure 4. We assume that procedure `foo` is called as `foo(100)` and that parameter `a` at line 1 is tainted with taint marking t_a. For ease of presentation, we discuss the example at the source-code level.

At `foo`'s entry, set S is empty. When line 3 is executed, our approach would recognize this as a conditional branching statement and add a pair $< 7, \{t_a\} >$ to S because t_a is the (only) taint marking associated with the branch's source operands. The next line executed is line 4. Because S is not empty, the taint markings in S are combined and associated with the statement's destination operands. In this case, there is only one set in S, and the resulting set of markings is $\{t_a\}$, which gets associated with variable `x`. Next, line 9 is executed, which corresponds to $ipdom(7)$. Pair $< 7, \{t_a\} >$ is therefore removed from S, which becomes empty. At `foo`'s exit, `x`'s set of taint markings is $\{t_a\}$, which is the correct result.

To account for situations like the one shown in Figure 2(b), where the above approach would miss the fact that `x`'s value depends on `a`'s value, we leverage a solution proposed in previous work (e.g., [13, 15]) of identifying each conditional instruction such that different memory locations are defined along the two branches of the instruction. For each such instruction, our approach adds, at instrumentation time, spurious definitions that make the set of memory locations defined along the two branches equal. For the code in Figure 2(b), this would be analogous to add a statement `x = x` on the (now empty) else branch of the statement at line 4.

The presented approach is safe under the assumption that we can analyze the binary code on which we are performing dynamic tainting and conservatively (1) build CFGs for the procedures in the code and (2) identify which memory locations are accessed by each instruction. Unfortunately, this assumption is often unmet due to the inherent difficulties in analyzing binary code, especially in the presence of indirect branches and indirect memory accesses. Suitably handling these issues requires dynamic updating, complex and often overly conservative analysis techniques [2], or additional knowledge about the compiler used to create the binary [4]. To simplify our initial implementation of the approach, we are currently limiting our tool to statically identifiable memory locations, similarly to Masri and colleagues [13].

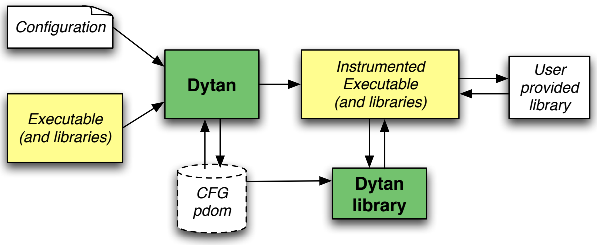

Figure 5: Overview of the DYTAN tool.

Specificities of x86 code. The x86 architecture supports conditional programming constructs, such as `if` statements and `for` and `while` loops, by means of a test-and-branch idiom: first, the program executes a test instruction, such as `cmp`, which sets some bits in the `%eflags` register; then, a conditional branch instruction is executed, whose outcome depends on the bit values in the `%eflags` register. Because all conditional branch instructions rely on the `%eflags` register, a propagation policy that does not account for implicit operands cannot implement control-flow based propagation. As discussed in Section 4.2.2, our approach considers implicit operands and suitably associates taint markings to the `%eflags` register, which enables safe data-flow and control-flow based dynamic tainting.

Another important aspect of the x86 instruction set is the difference between direct and indirect branches. Direct branches specify their target address as a constant memory address, while indirect branches use a register. For example, instruction `jmp 0x8048345` is a direct branch to address `0x8048345`, whereas `jmp (%eax)` is an indirect branch to the address stored in register `%eax`. For direct conditional branches, which do not have source operands, it is enough to consider the taint markings associated with the `%eflags` register. Indirect conditional branches, however, have source operands. The taint markings for such operands must be included in the set of taint markings that are propagated through control-flow (i.e., they must be added to the set of taint markings added to set S when the corresponding indirect conditional branch is executed). Our approach suitably handles both cases.

4.3 The Tool: DYTAN

We implemented our framework in a prototype tool called DYTAN (DYnamic Taint ANalyzer). Figure 5 provides an overview of DYTAN's mode of operation. To provide a friendly interface for the tool, we give users the ability to specify dynamic taint techniques through a *configuration* file in XML format. The user-provided configuration file specifies the kind of dynamic taint analysis to be performed in terms of taint sources, propagation policies, and taint sinks. Based on this configuration, DYTAN instruments the x86 *executable* on the fly to produce an *instrumented executable*. To perform instrumentation on the fly, DYTAN leverages the PIN dynamic instrumentation framework [12], which is well supported and offers a rich APIs for manipulating x86 binaries. Performing control-flow based taint propagation requires DYTAN to compute *CFG*s and *pdom* information at code loading time. To reduce the overhead, this information is computed only once per binary object and then stored along with a checksum of the object. When a binary object is loaded, DYTAN checks whether there are CFGs and postdominance information stored for that object and with the right checksum (the object could have been updated after the information had been stored). If so, it loads the information. Otherwise, it computes and stores it. While running, the instrumented program needs to access the DYTAN *library*, which provides taint-propagation functionality, and the *user-provided library*, which contains the checking operations associated with taint sinks and, possibly, custom operations for combining taint markings, as discussed in Section 4.1. We provide an example of DYTAN's usage in Section 5.1, where we use the tool to implement two different techniques based on dynamic-taint analysis.

Although we attempt to expose most of the framework's options through the XML configuration file, we also support power users that may need more flexibility than what the configuration file can offer. DYTAN provides direct access to its functionality through a C++ API. Using this API, users can register call back functions that implement their custom approach for marking, propagating, and checking taint markings. To use the API, users would write a C++ function with a predefined name that invokes the API methods to set configuration parameters and register functions in the user-provided library. Functions that implement a specific functionality must provide a predefined signature, which lets DYTAN substitute them to its internal default implementation of the corresponding functions.

We stress that DYTAN, by working at the binary level, can transparently handle external and system libraries. The tool simply instruments application and library code on the fly, at code loading time, and propagates taint markings appropriately.

5. EMPIRICAL EVALUATION

Our empirical evaluation has two main goals: (1) assess the suitability of our framework for implementing different types of dynamic taint analyses and (2) study how the specific characteristics of the dynamic tainting approach used can affect efficiency and accuracy of the taint analysis. We state these goals in terms of the two following research questions:

RQ1: Can we implement existing dynamic taint analyses with limited effort using our framework?

RQ2: To what extent the way information flow is handled affects the results and performance of taint analysis?

We first investigate these two research questions, and then present a small case study in which we measured the performance of our tool in terms of time and space overhead.

5.1 Research Question 1

To answer RQ1, we need to demonstrate that DYTAN can be used to implement different dynamic tainting techniques with a limited amount of effort. To this end, we selected two techniques previously presented in the literature and implemented them using DYTAN. To check that our implementations are faithful replicas of the original techniques, we partially replicated or emulated the empirical studies used to evaluate such techniques (which also serves as a sanity check for DYTAN).

5.1.1 Implemented Techniques

Prevention of overwrite attacks. The first technique that we re-implemented is the technique for preventing overwrite attacks presented in [17] and [21] and summarized in Section 3.2. We chose this technique because it is well known, clearly defined, has been implemented several times, and has been evaluated against a set of freely-available attack benchmarks, which allows us to replicate the original studies and, thus, assess how well our implementation reflects the original technique. Within our framework, the technique can be specified as follows:

```
<dytan-config>
<sources>
  <source type=''network''>
    <host>*</host>
    <port>*</port>
  </source>
</sources>
<propagation>
  <dataflow>true</dataflow>
  <controlflow>false</controlflow>
</propagation>
<sinks>
  <sink>
    <id>36</id>
    <location type=''instruction''>
      <instruction=''ret''/>
       ...
      <instruction=''jmp''/>
    </location>
    <action=''validate-absence''/>
  </sink>
</sinks>
</dytan-config>
```

Figure 6: Sample DYTAN configuration file.

Sources. The taint sources for this technique consist of any data that is read from the network. These data should all be tainted using a single taint marking.

Propagation policy. A propagation policy based on data-flow alone is sufficient in this case.

Sinks. There must be a sink for each instance of a call, return, or branch instruction. At each of these points, the target of the control transfer instruction should be checked to make sure that it was not tainted by data read from the network.

To provide an example of DYTAN's usage, Figure 6 shows how the informal description above would be encoded in DYTAN's configuration file. In this example, the network is the only taint source. The `host` and `port` tags allows users to indicate either a class of connections or an individual connection, with "*" being a wildcard that matches any host or port. Because no specific taint marking is specified, DYTAN would follow its default behavior and use a single taint marking. Therefore, according to the configuration, any data coming from any network connection would be tainted with a generic taint marking. The propagation section specifies that only data-flow based propagation should be used. Finally, the sink section specifies that the checking operation, function `validate-absence`, should be performed right before any of the listed instructions (the list should contain all control-transfer instructions).

To complete the implementation of the approach, we added to the user-provided library an implementation of `validate-absence` that checks whether the set of taint markings passed as a parameter is empty and terminates otherwise; when the check fails, it means that data from the network is determining the target of a control-transfer operation, which is a symptom of an overwrite attack.

Detection of SQL injection. The second technique that we re-implemented is a technique against SQL-injection proposed by Halfond and Orso [7]. It is based on (1) identifying trusted data in a web application (in most cases corresponding to hard-coded strings), (2) tainting such data, and (3) checking that all sensitive parts of the SQL queries generated by the application (i.e., everything except string and numeric literals) contain only tainted data. Because a taint marking indicates trusted, rather than untrusted, data, this technique is called positive tainting. We chose this technique because it is different in nature from the first one we selected and because it involves a more complicated dynamic taint analysis. For the sake of space, we do not show the XML configuration for this technique and just summarize its definition within our framework.

Sources. The taint sources consist of all hard-coded strings in the application, which should be tainted with a marking indicating complete trust. In addition, developers can specify additional markings to indicate, for instance, trusted data read from a file that should be treated in a specific way. For the study, we only considered hard-coded strings and specified them as memory locations in the binary code.

Propagation policy. Also this technique is only concerned with taint markings propagated through data flow. More precisely, the original definition of the technique focuses on a specific subset of data-flow propagation and only propagates taint markings associated to strings and that occur through string operations. We simply used the standard data-flow based propagation provided by DYTAN, which can conservatively propagate the taint markings associated with the hard-coded strings.

Sinks. We need a sink for each database access point (i.e., points in the code where a query is submitted to a database through a call to a specific function). The code location of the sink is the access point itself, and the variable of interest is the parameter that contains the query about to be submitted to the database. The checking operation consists of parsing the query and making sure that every character composing a SQL keyword or operator has a trust marking. If developers specified custom sources of trusted data, the checking function could be extended to handle data that contains these additional trust markings.

It is worth noting that the checking function for this technique is more complex than the one for checking code overwrites. It first uses an SQL parser on the string about to be submitted to the database (passed as a parameter by DYTAN) to identify tokens corresponding to SQL operators, keywords, and literals. Then, it checks the taint markings associated with the characters in the non-literal tokens. We were able to integrate the checking function from the original WASP tool into our implementation of the technique by (1) writing a wrapper around the original function that accepted the right parameters and (2) specifying the wrapper as the checking operation for the sinks in DYTAN's configuration file.

5.1.2 Evaluating the Two Techniques

To provide some confidence that the reimplementation of the two selected techniques generated by instantiating our framework is faithful, we partially replicated the studies performed to validate the original techniques. For the technique against overwrite attacks, we used the same benchmark suite developed by Wilander [24] and used in [21]. The benchmark consists of 18 different overwrite attacks that use a variety of exploits, including heap and stack overflows. Both our implementation and LIFT were able to prevent all 18 attacks. While not surprising, these results show that DYTAN allowed us to accurately implement, with low effort (see Section 5.1.3), the original technique.

For the technique against SQL injection, we could not reproduce the studies performed by the authors because the original implementation of the technique works on `JSP` servlets, whereas our implementation works on binaries. Therefore, we emulated the study by extracting several query-building sequences from the servlets used as subjects in [7] and converting them to equivalent C code that takes as input the same URLs used in the original study.

Table 1 shows the results of our implementation compared with the original results, in terms of number of attacks stopped and number of false positives generated. Also in this case, our implementation of the technique based on DYTAN achieved the same level of success at stopping SQL injection attacks as the original approach.

Table 1: DYTAN SQL injection false positive results

Subject	# Attacks	Successful Attacks	
		DYTAN	WASP
events	6209	0	0
checkers	4431	0	0
Subject	**# Legitimate Accesses**	**False Positives**	
		DYTAN	WASP
events	900	0	0
checkers	1359	0	0

5.1.3 Discussion of the Results

Overall, the effort required to implement the two techniques using DYTAN was fairly limited. Implementing the overwrite attack prevention technique took less than an hour. The only actual code we had to write was the checking operation associated with the taint sinks, which is a simple one. Implementing the approach against SQL injection was more complex. The most difficult parts were adapting and integrating the (existing) checking operation into the framework and finding a way to specify taint sources. The overall implementation time was still less than a day of work.

We are well aware that these results are preliminary in nature and highly qualitative, and that there are many threats to their validity because the users of the framework were also the ones who developed it. More extensive studies with external users are needed to address these threats and provide more confidence in the results. Nevertheless, we believe that the results for this first set of studies are promising, especially considering that our implementations were able to perform successfully on reproductions and emulations of the studies used to evaluate the original techniques.

5.2 Research Question 2

The goal of this part of the evaluation is to investigate the effects that imprecision can have on the results of taint analysis. Because more conservative taint-propagation approaches are typically also more expensive, it is important to assess whether the additional cost involves comparable benefits in terms of accuracy of the results. For some applications, unsafe results may be acceptable if they come at a much lower cost in terms of overhead imposed by the analysis. To investigate this issue, we performed dynamic taint analysis on two subjects using approaches with different degrees of conservativeness and compared the results obtained with the different approaches. More precisely, we considered the following approaches:

CF & DF is the approach corresponding to the control- and data-flow based propagation policy supported in our framework.

DF Full corresponds to the data-flow based propagation policy supported in our framework. It is conservative with respect to taint propagation that occurs through data flow, but disregards the effects of control dependences.

DF no IM refers to a data-flow only propagation that does not consider the effects of implicit operands (see Section 4.2.2).

DF no AG refers to a data-flow only propagation that does not consider the effects of address generators (see Section 4.2.2).

DF Direct is the least conservative of the five approaches. It accounts only for taint propagation that occurs due to data flow involving only explicit operands (i.e., it disregards the effects of indirect operands and address generators).

Throughout the paper, we provided various examples that showed how disregarding the effects of some code constructs could lead to loss of information and, thus, to unsafe results. To estimate the entity of such loss in practice, we used two real, widely-used software subjects: FIREFOX (`http://www.mozilla.com/firefox/`), a web

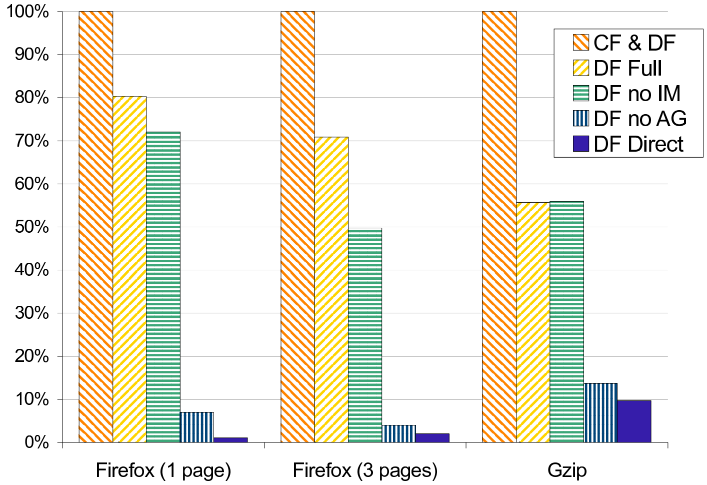

Figure 7: Tainting results for different propagations.

browser which consist of 850KB of binary code without considering shared libraries; and GZIP (`http://www.gzip.org/`), a compression tool which consist of 75KB of binary code without considering shared libraries. We used DYTAN to implement a straightforward dynamic taint analysis tool and ran it on the two subjects. The tool taints the programs' inputs (data from the network for FIREFOX and files to be compressed for GZIP) and dumps the taint information for the whole memory at the end of the execution. We generated five instances of the tool, each implementing one of the propagation approaches described above.

For FIREFOX, we considered two types of executions: one in which we loaded one page and one in which we loaded three different pages. In both cases, we exited FIREFOX right after the pages were loaded. For GZIP, we simply considered an execution in which we compressed a large file.

To collect the data, we ran each of the five versions of the tainting tool on FIREFOX and GZIP while executing them as described, and measured the number of memory bytes tainted at the end of the executions. We repeated the measurement ten times and averaged the results. Figure 7 shows the collected measures for the three kinds of executions: FIREFOX run on one page, FIREFOX run on three pages, and GZIP compressing a file. For each execution type, the figure shows the relative number of bytes tainted when each of the five taint-propagation approaches is used. 100% corresponds to the number of bytes for the most conservative approach (i.e., CF & DF). To better assess the relative differences shown in the figure, consider that the total amount of memory tainted for FIREFOX one page, FIREFOX three pages, and GZIP is 1.2MB, 2.6MB, and 2.2KB, respectively.

As the figure shows, there is a dramatic difference in the amount of memory tainted when using different taint-propagation approaches. It is apparent that not considering control-flow based propagation results in a considerable loss of information, ranging from 20% to almost 45%. The effect of implicit operands is relevant for FIREFOX but almost irrelevant for GZIP, whereas not considering address generators has a dramatic effect in all three cases, ranging from about 40% to more than 60% information loss.

One possible explanation for the difference in the effects of implicit operands and address generators is that address generators are usually used to calculate the location of a memory access. Therefore, disregarding address generators would result in many memory locations not being tainted with the generators' taint markings. Conversely, the most common implicit operand is the `%eflags` register, which is read almost exclusively by conditional branch operations. Therefore, when control flow is not considered, the effect of implicit operands is very limited. Our initial examination of the binary code of the two subjects and of the propagation logs optionally produced by DYTAN confirms this explanation.

Overall, the results provide a clear indication that disregarding some causes of taint propagation can have significant repercussions on the results of dynamic taint analysis and, therefore, on any application that relies on these results. Whereas for some applications the effect could be irrelevant, in other cases it may make the approach unsafe. Our framework, besides allowing for quickly implementing different dynamic tainting analyses, also lets users experiment with different variations of a specific technique and assess their relative effectiveness. When satisfied with the results for one variation, users could then keep using that instance of the technique or implement an ad-hoc, optimized version of the analysis with the same characteristics.

5.3 Performance of the Tool

To assess the time and space overhead imposed by DYTAN, we performed a case study using again GZIP. We did not perform the study on FIREFOX because its functionality is not CPU-bound, which causes the overhead to be masked by free cycles in the execution. To compute time overhead, we measured the time required to compress a file with GZIP for a normal execution and while performing dynamic taint analysis. We performed the same taint analysis used for our second research question, but considered only the two standard propagation policies provided by DYTAN: data-flow based and control- and data-flow based. Also in this case, we reran the measurements ten times and averaged the results. The time overhead we measured for data-flow based propagation alone was approximately 30x, whereas the overhead imposed by control- and data-flow based propagation was approximately 50x.

To calculate the space overhead imposed by DYTAN, we measured the memory allocated by GZIP during a normal execution and while performing dynamic taint analysis. Because GZIP is a batch program, we used an external program that took a snapshot of the memory when the compressed file being created reached a given size. As before, we averaged over ten measurements of the results. The resulting space overhead is approximately 240x.

These overheads are undoubtedly high, but they are comparable to the overheads reported in previous work on dynamic taint analysis (e.g., [3, 23]). One factor to keep into account is that those previous techniques were ad-hoc techniques, whereas in implementing our framework, we often had to trade time and space efficiency for flexibility. Moreover, our current implementation of DYTAN is unoptimized, and there is room for improvement. Currently, we have one set of taint markings for each byte; if contiguous memory locations have the same taint markings, as it is often the case, we can associate a single bit vector to the whole memory range. We also anticipate being able to reduce memory consumption by switching to a more efficient storage mechanism for taint markings, such as splay trees. The current version of DYTAN keeps all of the CFGs and postdominance trees for a program and related libraries in memory at once; we could easily optimize the tool by only loading graphs that are relevant for the part of code being executed. Analogously, there are several possible improvements that could speed up the taint propagation and reduce time overhead, such as using static-analysis information to avoid propagating taint markings that could never reach a taint sink or precomputing propagation within maximal basic blocks.

In short, although we do not expect this kind of framework to become as efficient as an optimized implementation that targets a specific task, we are confident that the overhead can be reduced by optimizing our implementation. Moreover, we are currently more interested in providing a general approach than an efficient one, and the overhead imposed was not a limiting factor in the preliminary studies that we performed.

6. CONCLUSION AND FUTURE WORK

We presented our generic framework for dynamic taint analysis, which provides several advantages over ad-hoc techniques and tools for dynamic tainting. First, it is highly flexible and customizable; specific taint analysis can be instantiated by simply specifying which data should be tainted and with which taint markings, how taint markings should be propagated during execution, and where taint markings should be checked and how. Second, it conservatively handles propagation of taint markings due to control- and data-flow. Finally, it works at the application level and can work transparently on programs that use external and system libraries.

We also presented DYTAN, a prototype tool that we developed and that implements our framework for x86 binaries. DYTAN leverages additional information possibly provided with the code, such as debugging symbols, but can perform dynamic taint analysis also on stripped binaries alone, which makes the tool widely applicable.

We used DYTAN to perform a set of preliminary studies. The first set of studies shows how DYTAN allows for implementing different dynamic tainting approaches with limited effort. The remaining studies illustrate how the different aspects of the taint analysis can affect its results. The studies also show the practical applicability of DYTAN, by running it on the `Firefox` web browser.

We have three main directions for future work. First, we will investigate ways to improve DYTAN's efficiency. We have several ideas of how the analysis can be made more efficient, some of which are discussed in Section 5.3. Second, we will gather feedback from users of the framework to assess whether our current framework needs to be extended to accommodate additional analysis (e.g., by performing tainting at the bit, rather than the byte level). Third, we want to use DYTAN to investigate specific applications of dynamic tainting in the context of software testing and debugging. In particular, we already started working on a debugging technique based on dynamic tainting.

Acknowledgments

This work was supported by NSF awards CCF-0541080 and CCR-0205422 to Georgia Tech and by the Department of Homeland Security and US Air Force under Contract No. FA8750-05-2-0214. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the US Air Force.

7. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI 90)*, pages 246–256, 1990.

[2] G. Balakrishnan and T. Reps Analyzing Memory Accesses in x86 Executables. In *Proc. Int. Conf. on Compiler Construction (CC 04)*, pages 5–23, 2004.

[3] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[4] C. Cifuentes Reverse Compilation Techniques. PhD Thesis: Queensland University of Technology, July 1994

[5] I. Corporation. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, 2006.

[6] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 13th International World Wide Web Conference (WWW04)*, pages 40–52, 2005.

[7] W. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, New York, NY, USA, 2006. ACM Press.

[8] J. Kong, C. C. Zou, and H. Zhou. Improving Software Security via Runtime Instruction-level Taint Checking. In *ASID '06: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 18–24.

[9] B. Korel and S. Yalamanchili. Forward Computation of Dynamic Program Slices. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 94)*, pages 66–79, 1994.

[10] L. C. Lam and T. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *22nd Annual Computer Security Applications Conference*, pages 463–472.

[11] T. Leek, G. Baker, R. Brown, M. Zhivich, and R. Lippmann. Coverage Maximization using Dynamic Taint Tracing. Technical Report TR-1112, MIT Lincoln Laboratory, 2007.

[12] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 190–200, 2005.

[13] W. Masri, A. Podgurski, and D. Leon. Detecting and Debugging Insecure Information Flows. In *International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 198–209, 2004.

[14] S. McCamant and M. D. Ernst. Quantitative Information-Flow Tracking for C and Related Languages. Technical Report MIT-CSAIL-TR-2006-076, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, November 2006.

[15] A. Moser, C. Kruegel, and E. Kirda Exploring Multiple Execution Paths for Malware Analysis. *IEEE Symposium on Security and Privacy*, May 2007.

[16] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[17] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[18] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *20th IFIP International Information Security Conference*, 2005.

[19] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID 2005)*, 2005.

[20] F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, 2002.

[21] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO '06: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148.

[22] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[23] 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 *MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[24] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[25] 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*, August 2006.