Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

# Development of a prototype taint tracing tool for security and other purposes

by

**Ulf Kargén**

LIU-IDA/LITH-EX-A--12/005--SE

2012-01-31


**Linköpings universitet**

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

# Development of a prototype taint tracing tool for security and other purposes

**by**

**Ulf Kargén**

LIU-IDA/LITH-EX-A--12/005--SE

2012-01-31

Supervisor: Nahid Shahmehri
Examiner: Nahid Shahmehri

# Abstract

In recent years there has been an increasing interest in dynamic taint tracing of compiled software as a powerful analysis method for security and other purposes. Most existing approaches are highly application specific and tends to sacrifice precision in favor of performance. In this thesis project a generic taint tracing tool has been developed that can deliver high precision taint information. By allowing an arbitrary number of taint labels to be stored for every tainted byte, accurate taint propagation can be achieved for values that are derived from multiple input bytes. The tool has been developed for x86 Linux systems using the dynamic binary instrumentation framework Valgrind.

The basic theory of taint tracing and multi-label taint propagation is discussed, as well as the main concepts of implementing a taint tracing tool using dynamic binary instrumentation. The impact of multi-label taint propagation on performance and precision is evaluated. While multi-label taint propagation has a considerable impact on performance, experiments carried out using the tool show that large amounts of taint information is lost with approximate methods using only one label per tainted byte.

## Acknowledgements

First and foremost I would like to thank my supervisor and examiner professor Nahid Shahmehri for her help and support, and for allowing me great freedom in shaping this thesis project according to my research interests.

I would also like to thank my opponent David Johansson for his very thorough review of my work, and for providing many useful comments and suggestions.

# Contents

# 1. Introduction

## 1.1. Background and motivation

As IT systems and computer software have come to play an increasingly important role in our everyday lives and to society in general during the last decades, software security has become an increasingly important concern. Finding software vulnerabilities is a slow and painstaking process that requires great skill. Modern techniques such as fuzzing can help find more software defects, but the results must be analyzed by a human in order to decide if the defect is a critical vulnerability that may be exploited by an attacker to gain access to the system, or just an obscure crashing bug that poses no direct risk of exploitation. Tools such as *!exploitable* [**1**] by Microsoft can provide a coarse assessment of the exploitability of a software defect, but much manual work is still required. The ideal solution would be to augment intrusion detection systems (IDS) with the ability to detect zero-day attacks[1] and automatically or semi-automatically create exploit signatures for the attack. This would require advanced software analysis tools that can provide detailed information on how input to a program affects its behavior. Such tools would also be useful to combat the growing threat of malware. Modern malware authors often utilize advanced obfuscation and anti-debugging techniques in order to prohibit or delay analysis of their malicious code by e.g. antivirus providers. Tools that analyze programs by looking at how they use their input rather than how the programs are written can overcome some of the obstacles posed by obfuscation techniques.

One approach to building such tools is so-called *dynamic taint analysis* or *dynamic taint tracing*[2]. Taint analysis is the process of analyzing how untrusted or "tainted" input to a program is used within the program. Static taint analysis of source code can be used to find software vulnerabilities, as in [**2**]. Dynamic, or runtime, taint analysis has been implemented for some interpreting languages. Such taint analysis can perform runtime detection of instances when untrusted input is used in an insecure way. One of the most well-known examples of this is the Perl *taint mode* [**3**]. Most programs in use today are however written in compiled languages such as C or C++ and the source code is in many cases not available to security researchers. Therefore, there has been an increasing interest in dynamic taint analysis of compiled (binary) programs in recent years. Proposed solutions to some of the problems mentioned above are among the suggested applications: Newsome *et al.* proposes a system based on dynamic taint analysis for runtime detection and signature generation of exploits [**4**]. Costa *et al.* proposes a similar system for containment of internet worms [**5**]. Yin *et al.* presents a full-system taint analysis tool for analysis of malware in [**6**]. Shahmehri *et al.* describes a system in [**7**][3] that utilizes dynamic taint analysis to automatically find vulnerabilities in executable files.

[1] I.e. an attack utilizing a previously unknown vulnerability.

[2] The terms *taint tracing* and *taint analysis* are often used interchangingly. In this thesis however, a distinction will be made between taint tracing, which is here defined as the process of tracing how taint propagates during program execution, and taint analysis, which is defined as the process of using taint information to deduce some new information. A taint analysis system requires some sort of taint tracing system for generating the taint information. This thesis is mainly concerned with the design and implementation of a taint *tracing* system, and not with any specific taint analysis application.

[3] At the time of writing, the paper has been submitted but not yet published. A preliminary version is available online.

Apart from security applications, another popular use of dynamic taint analysis being researched is automatic reverse engineering of unknown input formats. Such systems can partially reconstruct a description of an unknown input format of a program by analyzing how the program uses its input data. Examples of such systems are [**8**] and [**9**]. Another application is program testing. The COMET system described in [**10**] uses dynamic taint analysis to automatically increase test coverage. A related application is whitebox fuzzing for finding security vulnerabilities in programs, such as the system described in [**11**].

While most existing taint tracing systems are designed for a specific task, a general-purpose taint tracing tool could be a useful aid for general program comprehension, e.g. when debugging complex software. One example of such a tool is Flayer [**12**], which is a security auditing and debugging tool that utilizes simple taint tracing.

## 1.2. Goals

Most existing implementations of taint tracing systems sacrifice precision in order to improve performance. Some implementations, such as the tool Flayer mentioned above, only store one bit of taint information per memory bit, i.e. “tainted” or “not tainted”. More advanced systems such as Vigilante [**5**] labels each input byte, but store only one label per tainted byte. If the value of one byte is derived from multiple input bytes, some taint information is lost. For an IDS such as Vigilante, this means that exploits that depend on the attacked program deriving some value from multiple bytes of input cannot be properly detected. It is clear that in order to deliver accurate information on how input to a program affects its behavior in the general case, a taint tracing system must be able to handle taint for values derived from multiple input bytes properly. The goal of this thesis project has therefore been to develop a prototype tool for dynamic taint tracing of compiled executables that is able to record full taint information without the loss of precision introduced by the simplifications mentioned above. Since very little research has been published on the impact of such simplifications on precision and performance, another goal has been to use the tool to study this. Such investigation can also be used to decide which optimizations of the tool that are most worthwhile to implement.

While the goal of the project has mainly been to produce a prototype implementation for research purposes, the general ambition has been to keep performance of the tool within acceptable limits, so that analysis of “real world” applications, e.g. web browsers and word processors, is possible. The main constraint is on memory use, where the requirement is that the tool must not exhaust all available virtual memory on a 32-bit system[4] even when analyzing larger programs such as a web browser. The slowdown of the tool is less of an issue, at least if real-time performance of analyzed programs is not required. Slowdowns of more than approximately 2 orders of magnitude would probably make analysis of larger programs unwieldy however, and the aim has therefore been to keep the slowdown within this range.

The intended end result of the project has been a tool for performing taint tracing of x86 binaries, without the need for source code or symbolic information stored within the binaries. The tool should be able to analyze a program when it runs and later produce an output file that contains information on all instances where tainted data affected the program flow, including exactly which input bytes that were involved in each instance.

[4] On a 32-bit Linux system, 3 GB of virtual memory is typically available to regular processes.

## 1.3. Contraints

In order to be able to finish the project within the given time frame, the following constraints had to be introduced:

- The tool will only be able to analyze 32-bit Linux executables.
- The tool is only required to be able to perform taint tracing for normal x86 instructions. Support for floating point (x87) and SIMD[5] instructions such as MMX and SSE are considered optional requirements.[6]
- Preliminarily, the tool will only support tracing of input from regular files and standard input. Support for network sockets may be added later.

[5] SIMD (Single Instruction Multiple Data) is a parallel programming paradigm to allow the same operation to be applied on multiple values in parallel. Modern CPUs often provide a number of special SIMD instructions to speed up numerical calculations.

[6] This might seem like a large restriction. Most consumer applications however use relatively few floating point or SIMD instructions in the core program logic.

# 2. Theory

## 2.1. Dynamic taint tracing

Taint tracing is the process of tracing how tainted data propagates through a program as it executes. *Taint* is the property of data that it has been received from a so-called *taint source*. Sources of user supplied data are usually considered taint sources. When taint tracing is used for security applications, tainted data usually means either *untrusted* data or *sensitive* data. Typical taint sources are routines for reading data from a file, from a network connection or from direct user input. Taint tracing allows for analysis of how tainted data is used in the program. A location in a program that can receive tainted data is called a *taint sink*. Usually it is interesting to consider instructions that affect the program flow or routines that communicate with the outside world as taint sinks. This way, it is possible to analyze how input to a program affects its behavior. From a security standpoint, taint tracing can be used to detect if untrusted data is used in an insecure way. E.g., if unvalidated user input is used in a system call to the operating system that allows arbitrary commands to be executed, this might allow users to indirectly execute commands at a higher privilege level than was intended. Alternatively, taint tracing can be used to detect when sensitive data is leaked to an insecure channel, e.g. when the contents of a sensitive document being saved to a temporary file by a text editor.

Dynamic taint tracing is the process of performing taint tracing of a program during runtime. As mentioned in the introduction, this kind of functionality has been available for some interpreting languages for some time, but during the last few years there has been an increasing interest in dynamic taint tracing of compiled binary programs. This poses considerable greater challenges than taint tracing of programs written in interpreting languages. Some of these challenges will be explored in this thesis and common methods and solutions will be explained, with a focus on programs written for Intel IA-32 CPUs[7]. This chapter will provide the theoretical background to dynamic taint tracing, starting with the general theory and concluding with a discussion of specificities of dynamic taint tracing on x86 machines running Linux, which is the target platform for the prototype taint tracing tool described in later chapters.

## 2.2. Taint propagation

Taint propagation is the process of propagating taint appropriately as the program executes, and is the heart of a taint tracing system. Usually, tainted data initially resides in buffers in memory, where it was written by a taint source. When this data is used and new data is derived from it and stored elsewhere (at another memory address or in a register), the taint is propagated. In this thesis three principal types of taint propagation are considered; *direct*, *address* and *control-flow* propagation. Each will be described in the following sections.

One important consideration is to decide on the smallest unit of data that can carry taint. Some implementations, such as [**12**], consider the *bit* the smallest unit that can carry taint. The overhead of storing taint labels for each bit would however be too great when allowing an arbitrary number of

[7] The Intel IA-32 standard is frequently referred to as “x86” from the original CPUs implementing it (386, 486, etc.) and the two names will be used interchangingly in the text.

taint labels for each tainted data unit. In this work the *byte* is therefore considered the smallest unit of data that can carry taint.

Two terms used throughout the text that might require clarification are *overtainting* and *undertainting*. Overtainting refers to the case of false positives, i.e. that data is marked with taint labels even though it does in fact not depend on data associated with these labels. Undertainting is the opposite case, that data is not marked with taint labels even though it should have been. Overtainting is generally preferred, especially in security applications, as it gives a conservative estimation of taint propagation. There are however cases where overtainting becomes a problem and undertainting might instead be preferable (see section 2.2.3 below).

### 2.2.1. Direct propagation

This is the most intuitive propagation type, and relatively straightforward to implement. Direct propagation denotes the case where new data is somehow derived from tainted data, either via a plain copy from one location to another, or through an arithmetic or logic operation with two or more operands.

Most earlier publications on dynamic taint tracing have not discussed the implications of tracing multiple labels per byte. For the most part, implementing multi-label taint propagation is straightforward; operations that take multiple input operands simply propagate the union of the source labels to the destination. One detail that needs some especial attention is however how taint is propagated between bytes of a multi-byte data word. Consider for example the addition of two 32-bit words. Since the value of the first (least significant) byte might affect whether a carry bit is carried on to the second byte, the second byte must be considered dependent on both the first and second bytes of the input operands. The third byte will depend on the first, second and third bytes of the input operands, and so on. This means that taint can be propagated between bytes of a multi-byte word even when arithmetic is being performed with a hardcoded constant. The exact semantics of this multi-byte taint propagation varies between instructions, and taking the exact semantics into consideration would require a fair deal of extra processing when performing taint tracing. In this work, an approximate treatment has been introduced to handle these cases. The taint propagation mechanisms used for different kinds of instructions are summarized below:

- *Movement* instructions shall just copy (and replace) taint labels byte by byte, if e.g. the third byte of the source has taint labels *L*, then the third byte of the destination shall also have taint labels *L*.
- *Arithmetic* instructions, *shift* instructions and *bitwise rotation* instructions can cause the value of one byte of a multi-byte word to affect the value of the other bytes. The exact semantics of this kind of taint propagation depends on the specific operation and on the operands. The tool will use an approximation, where any instruction that falls into one of these three categories will cause the taint labels of *each* byte of the destination to be set to the union of the taint labels of *all* bytes of the input operands. Note that this approximation is conservative in the sense that it favors overtainting of some of the bytes in multi-byte words instead of losing taint information.
- *Logic* instructions (AND, OR, etc.) cannot propagate taint between bytes in a multi-byte word. Each byte of the destination shall have taint labels that are the union of the taint labels for the corresponding byte of the input operands.

In order to define the taint propagation rules in a uniform and compact way, the following special notation has been introduced:

> The set of taint labels for a data location, e.g. a register or a region of memory, will be denoted *T*. *T* will denote the set of taint labels for each byte of the data location. If the data location e.g. consists of four bytes, *T* will be a list with four elements, where each element will be a set of taint labels. The subscripts *D* and *S* will be added to *T* to denote destination and source data locations respectively. Sources and destinations are assumed to be the same size. The symbol Ø will denote a list of empty sets, one for each byte of the data location.
>
> The following operators will be used:
>
> - The assignment operator (=) will denote *bytewise* assignment of taint labels, as with a movement instruction.
> - The union operator (U) will denote an operation where *each* byte of the result will have a set of taint labels that are the union of the set of labels of *all* bytes in the operands. E.g. an arithmetic instruction will result in such a union.
> - The union operator with a "bw" subscript ($U_{bw}$) will denote a bytewise union of taint labels. Each byte of the result will have a set of taint labels that is the union of the taint labels of the corresponding byte in the operands. A logic instruction will result in this kind of union.

Figure 1 below shows an example of performing the two types of union operations.

Given the above notation, we can define the taint propagation rules for operations with one or two input operands and one output operand[8]:

1. *A constant value is stored in some location:* $T_D = \emptyset$ (This will be referred to as *untainting*.)
2. *A non-constant value is stored in another location:* $T_D = T_S$
3. *An arithmetic, shift or rotate operation is performed with a constant value:* $T_D = T_S \cup \emptyset$
4. *An arithmetic, shift or rotate operation is performed with a non-constant value:* $T_D = T_{S1} \cup T_{S2}$
5. *A logic operation is performed with a constant value:* $T_D = T_S$
6. *A logic operation is performed with a non-constant value:* $T_D = T_{S1} \cup_{bw} T_{S2}$

A special case is instructions whose result do not depend on the input operands. Examples of these are performing a bitwise AND with all zeroes, a bitwise OR with all ones or subtracting a value from itself. These cases should be treated as if a constant value was stored in the destination operand, i.e. an untainting operation.

[8] The most common case for x86 instructions is that instructions take two operands that are used as input and the result is stored in one of the operands. The taint propagation rules are however readily extended to include three or more operands if necessary.

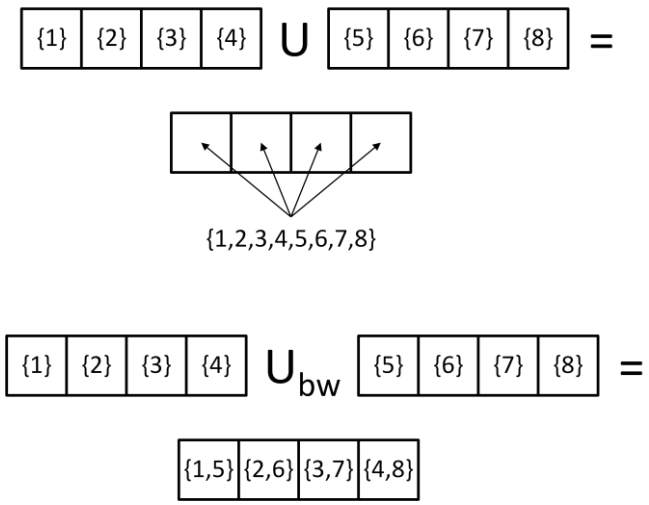


**Figure 1: Examples of "full" union (upper) and bytewise union (lower) taint propagation for four-byte values.**

### 2.2.2. Address propagation

Address propagation refers to the case where a memory region is accessed with indirect addressing using a tainted address. This shall propagate the taint labels of the *address*, regardless of whether the referenced memory itself is tainted or not. To describe the semantics of address tainting the following addition is made to the notation described in section 2.2.1:

> The subscript *A* will be added to *T* to denote the taint labels of data used as an address. The notation $T_{AU}$ is defined as a shorthand for $T_A$ U $T_A$. E.g., if addresses are four bytes, $T_{AU}$ will consist of a list of four identical sets, where each set is the union of all taint labels of all bytes of the address.

The following two rules describe the semantics of address taint propagation:

If a value is loaded from or stored into a location specified by a tainted address perform the following:

*a)* Adapt the size of the list $T_{AU}$ of taint label sets to the size of the source operand $T_S$ by removing or copying sets. Afterwards $T_{AU}$ will have the same number of taint label sets as $T_S$.

*b)* Assign taint according to: $T_D = T_S \cup_{bw} T_{AU}$

Figure 2 shows an example of address taint propagation.

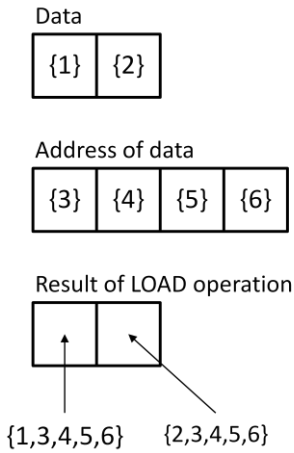


**Figure 2: Example of address propagation when loading a two-byte value using a tainted four-byte pointer.**

### 2.2.3. Control-flow propagation

Control-flow propagation is the most complex of the three modes. Control-flow propagation refers to the case where values of data locations are altered because of the control flow of the program. Consider the following example:

```
if(tainted_var == 0)
  a = 1;
else
  a = 2;
```

In this case the value of the variable `a` depends on the value of `tainted_var` even though it is not directly derived from it.

Control-flow tainting is difficult to implement and it is in the general case impossible to construct an algorithm that can perform complete control-flow taint propagation in all cases, since that would imply solving the halting problem. A brief overview of the method used in [**13**] to implement control-flow tainting is presented here; see the paper for full details.

In order to perform control flow tainting, all values that are altered due to a branch instruction should be marked with the taint labels of the operands to the branch instruction. When a conditional branch instruction is executed based on values of some tested operands, the taint labels of those operands are used to taint the branch instruction. All assignments that are executed because of that specific branch decision will be marked with those taint labels. Assignments that would have occurred regardless of which direction the branch took will not be tainted with the branch's taint labels. Formally, this is how control flow taint is treated:

- When a test is performed to decide if a branch should be taken, the union of all taint labels of all bytes of all input operands is calculated. This set of taint labels will then be the taint labels of the branch instruction, denoted $T_C$. Note that $T_C$ has logical size one byte, i.e. it represents only *one* set of taint labels. Apart from the input operands to the test, the destination address of the branch is in principle also an input operand of the branch instruction and should contribute taint if the branch is *taken*. However, since the method described here depends on statically generated control flow graphs (see below), branches to

run-time calculated addresses could not be handled. Jumps to tainted addresses can therefore not be considered during control-flow taint tracing[9].

- When a value is altered inside a conditional block with a tainted branch instruction, control flow taint is propagated as follows:

    1. First assign $T_D$ according to the rules of direct and/or address tainting.
    2. For each $T_C$ (in case of multiple nested tainted branch instructions)
        a. Extend $T_C$ to the size (number of bytes) of $T_D$ in the same way as with address propagation.
        b. Propagate taint according to $T_D = T_D \cup_{bw} T_C$

The algorithm for deciding which instructions that are conditionally executed because of a specific branch makes use of the Control Flow Graph (CFG) for the program. In a CFG, the nodes represent statements and the edges represent possible flow of control between statements. The algorithm is based on the principle of *postdominance.* A node *n* postdominates a node *m* in a CFG if all paths from *m* to the exit node go through *n*. *n* immediately postdominates *m* if there are no nodes in the path between *m* and *n* which are postdominated by *n*.

If *m* is a branch instruction, then all instructions between *m* and its immediate postdominator *n* are to some degree under the control of *m*, i.e. whether they are executed or not are partially dependent on the branch decision at *m*. The statement *n*, and all following instructions, is not dependent on *m* since *n* is executed regardless of the branch decision at m.

Consider the code listing and corresponding CFG in Figure 3. Here, the node corresponding to line 1 is a branch point. According to the definition above, nodes 5 and 6 are postdominators to node *1*, since all paths from *1* to the exit node go through *5* and *6*. Since there are no other postdominators in the path from *1* to *5*, *5* is an immediate postdominator to *1*.

The algorithm works by first generating a CFG by static analysis of the program to be analyzed. All immediate postdominators in the program's CFG are then identified. When an assignment takes place after a branch instruction, but before its immediate postdominator, the affected value is tainted with the taint labels of that branch instruction and all other preceding branch instructions whose immediate postdominator has not been reached.

Returning to the example in Figure 3, we see that according to the semantics of control-flow taint propagation the variable a should be tainted with the taint labels of `var`, while `b` and `c` should not, since they are not in the path between node *1* and its immediate postdominator.

[9] In x86 assembly, conditional branches can only have immediate (constant) target addresses, while unconditional branches (e.g. function calls) can have both constant and non-constant target addresses. This means that control-flow taint propagation due to e.g. function calls using runtime-calculated function pointers cannot be performed.

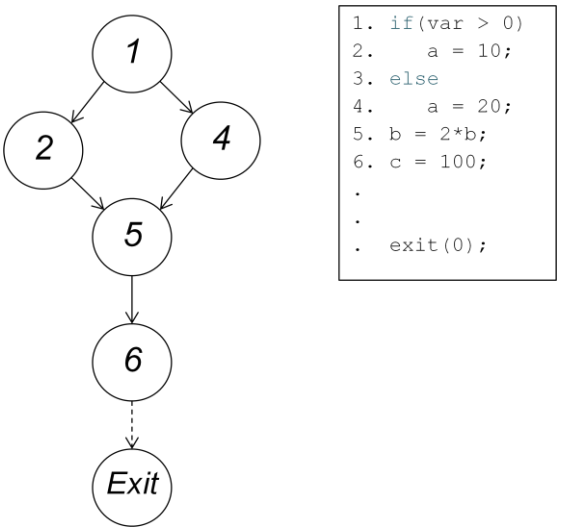


**Figure 3: Example of a control flow graph for a simple program snippet.**

Note that there could be both false positives and false negatives with this approach. Consider the following two cases:

```
(1) if(tainted_var == 0) {
      a = 1;
      b = 1;
    } else {
      a = 1;
      b = 2;
    }

(2) a = 0;
    if(tainted_var == 0)
      a = 1;
```

In the first case both variables `a` and `b` will become tainted, even though the value of `a` does in fact not depend on `tainted_var`. In the second case, `a` will not be tainted if the conditional statement is not executed, even though the only reason it still has the value 0 is because `tainted_var` was nonzero. It is possible to resolve the second problem by inserting a "virtual" else-statement that contains something like "`a = a`". Doing this however requires very advanced static analysis for all but the simplest cases, and in the general case this is an unsolvable problem.

Apart from the problems implementing control-flow taint propagation there are many subtle details that must be taken into consideration when performing control flow taint propagation to avoid severe overtainting. If a program for example performs various kinds of sanity checks of the input at the beginning, the taint of these conditionals will be propagated to virtually all data used by the program. An even more severe case of overtainting stems from the semantics of the `push` x86 instruction. This instruction first decrements the stack pointer by 4 (in a 32-bit environment) and then stores the value of its operand at the new address pointed to by the stack pointer. If this instruction were to be executed within a tainted control flow block, *both* the written memory area on the stack and the stack pointer itself would become tainted. Since the stack pointer may never become untainted again, almost every byte used by the program might eventually become tainted if address taint propagation is used. Cases such as these naturally result in huge performance losses

and could render the results of a taint tracing session useless. Using control-flow taint tracing in practice would therefore most likely require carefully crafted exceptions and application-specific tweaks for each individual case, making a generic control-flow taint tracing system unfeasible.

Because of the problems with control-flow taint propagation most of the existing approaches to dynamic taint tracing of compiled programs do not implement control-flow tainting. Note however that there are many relatively common cases of taint propagation in programs that cannot be captured without control-flow taint propagation. Consider for example the well-known `strlen` C-library function, that increments a counter in a loop until the terminating character of a string is found. Without control-flow tainting it is not possible to capture the property that the returned value depends on the input string. Some systems, such as Flayer [**12**], intercept calls to common library functions to handle cases like this, but the problem of how to achieve high-precision taint tracing in the general case without the problems introduced by generic control-flow taint propagation remains an open research problem.

## 2.3. Taint sinks

Taint sinks, as described above, are places in the program that are of interest to monitor in order to detect if they receive tainted input. What is relevant to consider as a taint sink differs depending on the application; in many security applications taint sinks are places in the code (function calls, etc.) that may pose a security risk if they are fed user-controllable data. Since the purpose of this thesis project has been to develop a generic taint tracing tool for program comprehension, a wider definition of taint sinks have been used. In the ideal case, all places in code that alters program flow or places where the program interacts with the "outside world" (i.e. the operating system) should be treated as taint sinks. Examples of these are:

- **Conditional branches.** This is the most obvious taint sink to monitor for program flow alterations. Both the input operands to the logical test and the target address[10] can carry taint into the taint sink.
- **Unconditional branches.** Only relevant if the target is a non-immediate value. Also affects program flow and could be interesting for security applications to e.g. detect or analyze return pointer overwrite attacks[11].
- **System calls.** System calls are the points where the program interacts with the operating system (and thus the outside world), and are interesting to monitor both for general program comprehension and security.
- **Regular function calls.** Calls to certain functions could be interesting to monitor as taint sinks. One such example is the standard C library memory allocation routines (malloc, free, etc.). If the input to such function calls is user-controllable this could pose a security risk, as a malicious user could e.g. cause a too short buffer to be allocated, resulting in a heap overflow later in the program.
- **Program crashes.** If a program crashes because of an illegal memory access, it could be interesting to be able to detect if an indirect memory access from a tainted source was the

[10] In case the target address is not an immediate (hardcoded) value.
[11] Note that e.g. function calls and function return instructions are just regular unconditional branches with some side effects.

cause. This could be used for e.g. analyzing the results of fuzz testing to detect various memory corruption bugs.

- **Execution of code in tainted memory.** Execution of tainted code is often an indication of a successful code injection attack[12]. This could be useful to monitor as a taint sink for e.g. exploit analysis and as a means of real-time exploit detection in intrusion detection systems.

## 2.4. Dynamic taint tracing on x86 Linux

While the previous sections dealt with the basic concepts of dynamic taint tracing in general, this chapter deals with some of the specific technical aspects of taint tracing on 32-bit x86 Linux systems. See [**14**] for details on the IA-32 (x86) standard.

### 2.4.1. Taint sources

System calls are typically the primary way a process can receive input from the user. Command line options and signals are other examples of input vectors into a program. On Linux, the most relevant system calls to consider as taint sources are those that deal with files and sockets. In order to implement taint tracing, these system calls must be intercepted and taint labels assigned to memory properly.

On Linux (and other UNIX systems) files are read using the `open` and `read` system calls. Files must first be opened by passing a file path to `open`, which on success returns a *file handle* that can be used to access the file. By passing the file handle and a pointer to a memory buffer to the `read` system call, the contents of the file can be copied into memory. To use files as taint sources, the `open` system call must be intercepted and its input parameters and return value must be examined in order to be able to associate future `read` system calls to the appropriate input file. The calls to `read` must also be monitored in the same fashion and taint labels assigned appropriately to the memory in the input buffer. Other system calls, such as `lseek`[13] and `dup`[14], must also be monitored and handled properly. Apart from the well-known ones mentioned here, there are also several other system calls for working with files. A taint tracing system must monitor all of these calls and be aware of their semantics.

Similarly, to treat sockets as taint sources the `socketcall` system call must be monitored to detect the creation of new sockets. Sockets are read from using a returned file handle passed to `read`, just like when reading from a file.

Command line options passed by the user to the program are simply stored on the stack and passed as arguments to the programs main function (e.g. “main” in C/C++); the taint tracing framework thus only needs to mark the memory on the stack that holds the command line with appropriate taint labels. Signals cannot carry any user-controllable input into the program and are thus usually not very interesting to treat as taint sources.

### 2.4.2. Taint propagation

Direct propagation of taint is performed in practice by monitoring all executed instructions and propagating taint according to their semantics. Apart from the regular instructions for integer

[12] See e.g. Aleph One’s well known Phrack article [**26**] for a prototypical example of a return pointer overwrite and code injection attack.
[13] Used to change the current reading position in a file.
[14] Used to create an alias for a file descriptor.

arithmetic the IA-32 standard contains a large number of special instructions, many with relatively complicated semantics, for accelerating numerical calculations. Examples are the x87 instructions for IEEE 754 floating point calculations and various SIMD instructions for parallel numerical calculations (MMX, SSE, etc.). These special instructions are used heavily in software for scientific computations, 3D rendering, etc., but "regular" consumer applications usually contain relatively few such instructions[15]. It is therefore a reasonable simplification to disregard these instructions when performing taint propagation.

When performing direct taint propagation it is necessary to pay attention to instructions that always result in the same output value regardless of the input operands (see section 2.2.1). Most such instructions are very rare in consumer software, since compilers usually don't generate such code. One important exception in x86 assembly is the "`xor reg,reg`" idiom for zeroing out a register. Compilers very frequently use this method instead of explicitly loading the value zero into the register, since the XOR method results in faster and more compact code. Another such method, that is sometimes used by compilers, is performing an OR with all ones (`or reg,0xFFFFFFFF`) in order to set all bits of a register to one.

Another important aspect of direct taint propagation is proper propagation of taint through system calls. Some taint tracing methods don't allow direct monitoring of code running in kernel mode (see section 3.2). These methods must rely on passive monitoring of input and output parameters of system calls, and be aware of their semantics. System calls in Linux receive their input parameters through registers and put the return value in the EAX register. Some system calls can also read or write memory specified by pointer arguments from the caller. Optimally, taint must in all cases be properly propagated to/from memory and registers when a system call is made.

On x86 machines, address propagation can occur both trough explicit loads or stores, or indirect memory access in various instructions. This kind of taint propagation is performed by monitoring instructions where the value of a register is used as a memory address and propagating taint from this register appropriately.

Control-flow taint propagation is, as mentioned above, restricted to branches to statically known addresses. Conditional branches in x86 always have immediate targets, so all conditional branches can in theory be considered. One challenge is keeping track of the taint from the input parameters of the logical test of the branch instruction; see the section on taint sinks below.

### 2.4.3. Taint sinks

This chapter provides a brief discussion of some specifics of handling the six taint sink types mentioned in section 2.3 in an x86 Linux environment:

- **Conditional branches.** In x86, conditional branches are implemented in such a way that the logical test and the actual branch are two separate instructions. Most arithmetic or logic instructions result in some status flags being set in a special register called EFLAGS. These flags indicate properties of the result, e.g. if the result is zero, negative or positive, or if an overflow occurred. There are also specialized instructions for setting these flags, which perform some computation and throws away the result. The actual conditional branch

[15] This is a generally accepted "truth" among people working with binary auditing. See e.g. [**27**].

instruction follows after the test instruction and uses the flags to decide whether to perform the branch or not. Note that the branch instruction must not necessarily follow directly after a test instruction. To properly handle the input operands of conditional branches, the taint tracing system must be able to remember which instruction that last affected the EFLAGS register. The EFLAGS register must also be able to carry taint. When a branch occurs, this information is recorded as a part of the taint sink event. In x86 the target address of a conditional branch must be an immediate value, which means that only the conditional can carry taint into the branch.

- **Unconditional branches.** In x86 assembly, unconditional branches can use both immediate values and registers for specifying the target address. In the latter case, taint associated with the register must be recorded in a taint sink event.
- **System calls.** As mentioned in the preceding section, both registers and memory can convey taint into a system call. All system calls must be intercepted and their input must be checked for taint.
- **Regular function calls.** Implementing a generic way to use functions as taint sinks in compiled programs poses a considerable challenge. If symbolic information is available for all functions in the executable, it is possible to check if the target address is associated with the (symbol) name of a function for each `call` instruction, and match that name to a list of monitored taint sinks. Many consumer products however have their symbolic information stripped before distribution to save space. In this case, it might still be possible to detect calls to certain functions by checking the target address against a list of known addresses of certain libraries. This method is of course only applicable to functions contained in well-knows libraries. A drawback of this method is that the database needs to be aware of all common versions of popular libraries, and also be updated as soon as a new version of a library is released. The method also does not work at all if executables are statically linked, rather than utilizing shared libraries. Cristina Cifuentes describes a technique in [**15**] that can partially overcome this problem. By creating *signatures* of functions, e.g. by calculating a hash of the first instructions in each function, it is possible to distinguish known library functions even when they are statically linked into the code. This method of course shares the same drawbacks as the method of checking for known function addresses.
- **Program crashes.** Crashes due to illegal memory access can occur when a register is used to address memory. As mentioned in the preceding section, this can occur either in an explicit load or store instruction, or due to indirect memory access in some other instruction. To treat such crashes as taint sinks, all instructions need to be monitored during execution, and a preliminary taint sink event needs to be recorded *before* each instruction that uses a register to address memory. As soon as such an instruction is detected, the old preliminary taint sink event is thrown away and a new is created. When an illegal memory access is detected in Linux, the offending program is sent a SIGSEGV (Segmentation fault) signal. The taint tracing framework needs to intercept SIGSEGV signals before they are sent to the program, and record the last preliminary taint sink event as a regular taint sink event. Since the program is interrupted immediately by the operating system upon an illegal memory access, the last preliminary taint sink event is guaranteed to be associated with the register holding the illegal address that caused the crash.
- **Execution of code in tainted memory.** A basic approach to detecting execution of tainted instructions would be to perform a check each time the instruction pointer (EIP) was updated

and record a taint sink event if the memory pointed to was tainted. Doing this for every instruction would however result in a huge performance penalty. A better approach would be to check the memory at the target address of each executed branch instruction for taint.

# 3. Implementation

Several approaches to dynamic taint tracing of compiled software have been suggested. Some methods, such as [**16**], relies on special hardware to perform dynamic taint propagation. Such methods typically offer superior performance to other approaches, but the need for special hardware prohibits widespread usage. A related approach is to use special virtual machines, which offer taint tracing functionality in the simulated hardware. An example is the system by Yin *et al.* [**6**] for analyzing malware. The benefit of using such methods is that full-system taint information can be recorded, i.e. taint propagation between processes. Another common approach, that does not require an entire separate virtual environment, is *dynamic binary instrumentation* (DBI). DBI frameworks work by instrumenting programs during runtime to record information about the program execution. Examples of systems based on dynamic binary instrumentation are [**4**], [**5**], [**8**], [**17**] and [**18**].

## 3.1. Dynamic Binary Instrumentation

The basic idea of dynamic binary instrumentation is to interleave the original code of the program being analyzed with analysis code that record information about program execution. When the program executes the analysis code is added in runtime and executes along with the original instructions. DBI frameworks offer this instrumentation functionality as well as the means for writing analysis tools. Typically such frameworks offer an API to inspect the original code of the program being analyzed and add appropriate analysis code. It is important to notice the difference between processing that happens during *instrumentation* and processing that happens during *runtime*. When a block of code of the analyzed program is to be executed, the DBI framework passes this code to the tool's instrumentation routine, which inspects it and inserts analysis code that is executed together with the original program code. Typically, DBI frameworks provide a cache of instrumented code, so that instrumentation only needs to be done once for most code blocks. Since much code in a program is executed many times in loops, the instrumentation code can be more complex than the runtime analysis code, which needs to be as fast as possible to reduce overhead. Note that the instrumentation routine in the tool only has access to the static code of the analyzed program, while the added analysis code has access to the current runtime state of the program.

Pin [**19**], DynamoRIO [**20**] and Valgrind [**21**] are three well-known examples of DBI frameworks. In [**21**] Nethercote *et al.* classifies Valgrind's instrumentation method as *disassemble-and-resynthesize* (D&R) while Pin and DynamoRIO are classified as *copy-and-annotate* (C&A). In C&A the original instructions of the analyzed program is copied through verbatim to the instrumented version of the code. The framework provides the instrumentation routine of the tool with annotations, which describe the effect of each instruction. Using these annotations, the tool adds analysis code. In D&R the original code is disassembled and translated into an *intermediate representation* (IR). The tool inspects the IR representation of the original code and adds analysis code, also expressed in the IR. The DBI framework then recompiles the instrumented IR back to native code so that it can be executed. One problem with C&A, which is avoided by D&R, is conflicts between the original code and the analysis code. If the analysis code e.g. uses some register that is also used by the original program code, the register needs to be saved to memory and later restored. Such conflicts must be handled, either manually by the tool writer or automatically by the DBI framework in C&A, while D&R handles this implicitly in the recompilation process. Another benefit of D&R is that a tool can

potentially work on multiple platforms without modification of the source code. The main downside of D&R is that the extra cost of translation between IR and native code usually results in greater overhead than C&A.

Valgrind uses a RISC-like IR called VEX in which all side effects (e.g. status flag changes) are made explicit. When writing instrumentation code for CISC-architectures like x86 this is a great benefit, since the IR is simpler and contains fewer instructions than native x86 code. Valgrind also contains advanced features for e.g. monitoring system calls and memory allocations. Because of the ease of tool development offered by the IR representation and some powerful features not present in other DBI frameworks (see the following section), Valgrind has been chosen for implementing the prototype taint tracing tool in this project. The following section provides a brief introduction to Valgrind.

## 3.2. Introduction to Valgrind

Valgrind is an open source DBI framework specifically designed for what the authors call "heavyweight" dynamic binary analysis tools. The use of a platform independent IR allows for advanced instrumentation and tools that work on multiple platforms without changing the tool's source code. Valgrind is available for multiple processor architectures (x86, Power PC and ARM) and supports both 32 and 64 bit x86 and PowerPC programs. Linux is supported on all processor architectures and Mac OS X is supported on x86/AMD64. Windows is not supported. This section provides a brief introduction to Valgrind, with an emphasis on the properties and features that are most important for the implementation of the taint tracing tool. The sources of the information in this section is [**21**], the documentation available at the Valgrind web page [**22**] and the Valgrind source code, also available at the web page. See these sources for complete details.

As mentioned in the preceding section, Valgrind translates the analyzed program into an intermediate representation called VEX. VEX has some RISC-like features, such as using a LOAD/STORE architecture (i.e. all memory read/writes are explicit). It also uses virtual pseudo-registers called *temporaries* which are *single-static-assignment* (SSA), i.e. a temporary can only be assigned a value once. When the IR is compiled to native code these temporaries are assigned real registers or spilled to memory, much like variables in a regular compiled language. Reads and writes to real registers are made explicit in the IR with the "GET" and "PUT" instructions. The entire state of the native CPU is stored in a special data structure in memory that is updated by e.g. PUT instructions. Valgrind also provides two copies of this data structure, which are also directly accessible from the IR. This allows for storing *shadow state* of e.g. registers; when the value of a real register is updated, the corresponding shadow register in a shadow state can be updated with some information about the new value of the real register.

Instrumentation in Valgrind is performed on single-entry multiple-exit *superblocks.* Each superblock contains the VEX representation of up to about 50 native instructions. Every superblock has its own set of temporaries, i.e. the scope of a temporary is its superblock. The instrumentation API is based on C and allows tool writers to register a callback function that is invoked before a new superblock is executed. The callback can inspect the superblock and insert analysis code, either as pure IR, or in case more advanced processing is required, as callouts to regular C functions. For performance reasons, an "inlined" IR implementation is preferable to a function call since the extra cost of setting up a stack frame and passing parameters on the stack can be considerable, especially for analysis

code that is executed frequently. Valgrind also serializes multithreaded programs in such a way that only one thread at a time executes and thread switches only occur at the end of a superblock. This is important when maintaining a shadow state; an update to the program state and the accompanying update to the shadow state must occur atomically to avoid inconsistencies.

One problem with DBI is that both the analyzed program and the instrumentation framework usually executes as regular user-mode processes. This means that the framework cannot directly monitor processing that occurs in kernel mode (i.e. in system calls). Valgrind solves this by providing system call wrappers that are aware of the semantics of almost all available system calls on supported systems. This means that Valgrind can provide detailed information to the tool about memory reads and writes in system calls. According to [**21**] this is a feature unique to Valgrind among all DBI frameworks. The information is made available to the tool via event callbacks. There are higher-level callbacks that provide information about the system call number and values of arguments, as well as lower level ones that are called each time memory or registers are read or written by a system call.

## 3.3. Implementation of the tool

The tool has been implemented according to the requirements and constraints in chapter 1. It can perform taint tracing of input from regular files and standard input according to the rules in section 2.2. Address propagation has been implemented but not control-flow propagation, due to the problems with control-flow propagation described earlier. The tool supports three types of taint sink types from section 2.3: unconditional branches, conditional branches and system calls. As stated in the constraints, all floating point and SIMD instructions are ignored for now, but instrumentation of these instructions could easily be added later on. All taint sink events are written to an output file in a binary format to keep file sizes low and allow fast parsing of files. The following sections provide some details about the implementation.

### 3.3.1. Taint sources

In order to trace taint from regular files, Valgrind's system call wrappers are used to intercept calls to the file handling system calls. The basic principle laid out in section 2.4.1 is followed to assign taint labels to read input bytes: The "open" system call is monitored to keep track of all open files and their file descriptors. Subsequent reads from these file descriptors results in taint labels being assigned to corresponding bytes in the file. A special data structure is maintained to hold mappings between file positions and taint labels. Other system calls such as `dup` and `lseek` are also monitored to allow correct mapping between file reading positions and taint labels. Currently, only read-only files are considered for taint tracing. It would be possible to handle read/write files, but then all "write" (and related) system calls must also be monitored to keep track of the current file position. The issue of how to handle the taint status of data that is first written to the file and then read back also arises when tracing taint from read/write files. Standard input is handled as a special case of regular files; the file descriptor with value 0 is always added to the list of open file descriptors at program startup if tracing of standard input is enabled.

### 3.3.2. Taint propagation

Taint labels are represented as 32-bit integers, which allows about 4 billion unique taint labels. Sets of taint labels are represented using simple sorted linked lists to simplify implementation and minimize memory usage. Shallow copies and reference counting is used when copying taint label sets to improve performance and reduce memory usage. There are three principal locations in which a

tainted byte can reside: memory, (native) registers or temporaries. Each of these locations must have a corresponding shadow-location to store the set of taint labels for each byte. Shadowing of memory and registers is relatively straightforward: Shadow memory is organized in a two-level table similar to a page table and shadow registers are stored in one of Valgrind's shadow states. Shadowing of temporaries is however a bit more involved. The approach used by the well-known tool Memcheck [**23**], which is also based on Valgrind, is to allocate another shadow temporary to hold shadow data for each temporary. This approach turned out to not be feasible for this project due to the large amount of temporaries needed for storing pointers to linked lists for every byte of every temporary. Instead, a static pool of memory is used. Space in the pool is allocated during instrumentation time and used by the analysis code to propagate taint during runtime. Special code is inserted to clear the pool after each superblock is finished. This approach works because Valgrind serializes execution so that only one superblock at a time uses the pool.

Taint propagation is performed by inserting callouts to helper functions. This is sub-optimal from a performance perspective, but necessary since sorting labels into linked lists and handling reference counting is simply too complex to implement in the VEX IR. As explained in section 2.4.2, taint must also be propagated to status flags in order to e.g. be able to record taint sink events for conditional branches. Since all side effects are made explicit in the IR, this requires no special effort and is implemented using the mechanics described above for propagating taint to registers and temporaries.

Taint propagation through system calls is implemented using Valgrind's low-level system call wrappers. The implementation does not consider the exact semantics of the system call, but simply records the union of all taint labels in registers and memory read by the system call and propagates this taint to all registers and memory written by the call.

### 3.3.3. Taint sinks

The tool maintains a list of taint sink events in chronological order. A taint sink event contains the following data:

- Information about the type of event (unconditional branch, conditional branch or system call).
- The union of all taint labels received by the taint sink.
- The code address where the event occurred
- The current stack trace.

In case of conditional branches both the address of the branch and the address of the test is stored. For system calls, a separate list of taint labels for each argument is recorded in the taint sink event.

Conditional and unconditional branches are recorded using instrumentation, just like regular taint propagation. For each branch, analysis code is inserted that calls a function to check if its operands are tainted, and records a taint sink event if that is the case. System call taint sink events are recorded using Valgrind's low-level system call wrappers, and works much like the taint propagation mechanism for system calls: When a system call happens, the taint of all registers and memory read by the call is recorded in the taint sink event.

One particular challenge is to record correct stack traces. The actual taint sink events often occur in library functions, so having correct stack traces is vital in order to be able to see where in the actual analyzed program the event occurs. Valgrind has built in support for walking the stack and retrieving a stack trace in the classical manner. This however does not work when the analyzed program or some of its libraries don't use the frame pointer. If debugging information is present in the program, this can be used by e.g. some debuggers to retrieve correct stack traces even without the frame pointer enabled. Support for using this kind of debug information is however not built into Valgrind, and the problem of how to handle programs without debug information still remains. To tackle this problem an optional functionality has been implemented, which dynamically keeps track of the call stack by instrumenting all call and return instructions. This can sometimes help to create better stack traces for programs that do not use the frame pointer, but fails to properly handle non-standard call and return sequences, which can also sometimes lead to incorrect stack traces.

### 3.3.4. Output

When the analyzed program exits, all taint sink events are written to file in a binary format. The tool can also handle the case of an analyzed program crashing and still produce an output file with all taint sink events up to the point of the crash. A trace file contains a string table with names of input files and executable images, a mapping between taint labels and byte ranges in the input files, a list of all stack traces referred to in taint sink events, and lastly the list of taint sink events in chronological order. A small Python program has also been implemented for parsing the output files and displaying all taint sink events in human-readable form.

### 3.3.5. User interface

The tool, like most tools based on Valgrind, uses Valgrind's built in support for creating command line interfaces. The default is to apply taint tracing to input from all files read by the program, but the user can specify exclusive filters to exclude files matching a certain pattern. Inclusive filters can also be specified. Files matching such filters will always be traced regardless of exclusive filters. It is also possible to specify which ranges of bytes in a file that should be traced. The user interface also allows choosing to enable or disable address taint propagation and taint propagation through system calls, and to choose which taint sink types to use.

# 4. Evaluation

In this chapter the tool will be evaluated with respect to performance and some alternative approaches to taint tracing will be discussed. The improvement of precision with the multiple-labels-per-byte approach of the tool, compared to the single-label approximation, is also discussed. The chapter is concluded with some suggestions on future work and improvements.

## 4.1. Performance

When discussing the performance of the tool there are two major sources of overhead to consider: The "baseline" overhead introduced by the DBI framework and the analysis code added during instrumentation, and the taint propagation overhead caused by the extra processing required to propagate taint. The first source of overhead is always present, since the analysis code is always executed even if there is no taint to propagate. It also depends less on the specific program, and is approximately the same for most programs. The second source of overhead depends heavily on program input and the specific processing performed by each program.

Performance of the tool has been measured for four programs; two "heavyweight" ones that utilize an advanced GUI, namely a web browser (Firefox 3.6.22) and a word processor (OpenOffice Writer 3.2.0), and two more "lightweight" non-GUI programs; the GNU C compiler (gcc 4.4.3) and the lightweight console-mode web browser Lynx (version 2.8.8). Each program was started with a specific input file and the startup times of the programs were measured. For Firefox and Lynx a 3.2 kB HTML file was used as input, for Open Office Writer a 7.8 kB ODT-file was used and gcc was executed on a 2.9 kB C source file. Three measurements were taken; the startup time with no analysis (native execution), the startup time when running the programs with the taint tracing tool but with no input defined as a taint source, and finally the startup time when considering the files described above as taint sources. The startup time was measured by extracting the total CPU time (i.e. the time actually spent executing on the CPU) in both user and kernel mode for the programs from the Linux *proc* file system. For the interactive programs, the CPU utilization was continuously monitored and when it stabilized at an idle value (well below 100%) the program was shut down with a termination signal and the total CPU time was recorded. For gcc, which is the only non-interactive program among those tested, the total CPU time of the compiler process was measured. This was repeated multiple times and an average execution time was calculated. The peak virtual memory use for each invocation was also recorded. The tool was configured to use unconditional branches and system calls as taint sinks. Conditional branch taint sink events were not used, since this leads to a huge amount of taint sink events being generated when tracing taint from large input byte ranges. Address taint propagation and propagation through system calls were also disabled during the measurements, as these taint propagation methods can sometimes lead to "taint explosions". Only pure data-flow based taint propagation were thus considered. The results are presented in the following section.

Due to very long startup times for the "heavier" programs when performing taint propagation, only 5 measurements each were performed for this case. For the two "lighter" programs 10 measurements were performed. The lightweight programs also presented a difficulty when measuring their native startup time, since their execution times were sometimes within the same order of magnitude as the minimum measurable time unit of CPU time. For these cases, a very large number of execution times

were measured (about 100) and the average was taken. The measurement precision for these programs is however still fairly low.

The low number of measurements of course limits the precision of the results, but since the execution times vary widely for different programs, or different input to the same program, taking exact measurements for these specific programs are not of great interest. Instead, these performance benchmarks serve to provide a coarse estimate of what overhead to expect from the tool.

The choice to use the peak virtual memory allocation as a measure of memory consumption also leads to some imprecision, especially for programs with low memory footprint, as discussed below. Again, the argument for doing so is that exact measurements are not of great interest for the purpose of this evaluation, and that the peak virtual memory gives a good indication of the memory overhead while still being simple to measure.

### 4.1.1. Results

The average execution times and slowdowns relative to native execution are presented in Table 1 below. Average maximum memory consumption for the same test suite is presented in Table 2. The entire results of all benchmark runs of the taint tracing tool can be found in Appendix A. (The benchmarks of native execution have been left out to save space.)

| | Native startup time (s) | With taint tracing tool, no taint | | With taint tracing tool, with taint | |
|---|---|---|---|---|---|
| | | Startup time (s) | Slowdown | Startup time (s) | Slowdown |
| **Firefox** | 1.67 | 481.28 | 288 | 3694.67 | 2212 |
| **OpenOffice Writer** | 0.91 | 259.30 | 285 | 766.48 | 842 |
| **gcc** | 0.093 | 19.73 | 212 | 19.91 | 214 |
| **Lynx** | 0.019 | 9.48 | 499 | 10.08 | 531 |

**Table 1: Average execution times and slowdowns relative to native execution for the four programs.**

| | Native memory use (kB) | With taint tracing tool, no taint | | With taint tracing tool, with taint | |
|---|---|---|---|---|---|
| | | Memory use (kB) | Relative increase | Memory use (kB) | Relative increase |
| **Firefox** | 177064.0 | 379736.0 | 2.14 | 432386.0 | 2.44 |
| **OpenOffice Writer** | 208034.0 | 419001.0 | 2.01 | 490347.0 | 2.36 |
| **gcc** | 2240 | 94756.0 | 42.30 | 94756.0 | 42.30 |
| **Lynx** | 8587.9 | 54508.0 | 6.35 | 54196.0 | 6,31 |

**Table 2: Average peak memory use for the four programs.**

As can be seen from Table 1, the baseline time overhead was between 200 and 300 times for all programs except Lynx, which had a higher overhead. The results for Lynx are however a bit unreliable due to the extremely low native execution time, which is difficult to measure as explained earlier. As expected, the runtimes when doing actual taint propagation varies heavily with the application and the input. For the more lightweight programs the baseline overhead appears to dominate over the propagation overhead, while for the GUI-driven applications the propagation overhead is dominant. The average values for the runtime of Firefox and OpenOffice Writer are a bit misleading; as can be seen in Appendix A, the execution time varies wildly between different invocations, with the lowest and highest execution times differing by over a factor 20 for Firefox. The corresponding memory use

also varies consistently, but less dramatically. This phenomenon is also seen in the benchmarks when using no taint, but the variations are considerably smaller. The likely explanation for this is that the programs don't behave exactly the same on each invocation, but sometimes does some extra processing. The very "heavy" processing required for taint propagation[16] can cause small differences in the behavior of a program to lead to drastic variations in runtime when taint tracing, depending on how much taint propagation that must be performed. This theory could possibly be verified by counting the total number of taint propagating instructions (see section 4.3) while performing the performance measurements. Since the overhead depends not only on the number of taint propagating instructions, but also on the amount of taint labels propagated and the semantics of the instructions, it could however still be difficult to directly correlate an increased number of executed instructions with an increased overhead.

The measured peak memory use, presented in Table 2, shows that the larger applications consume approximately twice as much memory in the baseline case, compared to the native execution case. This extra memory use is probably mostly due to the extra memory consumption caused by Valgrind's code translation. It can also be seen that the memory consumption don't increase drastically when doing actual taint propagation. There are a few additional peculiarities that require some attention: The memory use when taint tracing gcc with and without taint are exactly the same. This is because the values recorded here are the peak virtual memory allocated to the program by the operating system, not the amount of memory that was actually used. The memory allocator allocates memory from the OS in larger blocks, which are then split up in smaller chunks and delivered to the program by the memory allocation algorithm (e.g. malloc). This means that smaller variations in memory use might not be visible when only looking at the total virtual memory consumption. Another interesting observation is that the memory consumption for Lynx is actually *smaller* when taint tracing with taint than without taint. The reason for this is also most likely some peculiarities of the memory allocation algorithm used.

In summary, it can be seen that the memory consumption is not a very big problem, at least when doing pure data-flow based taint propagation. The runtime overhead however is very large. For the smaller non-graphical programs the overhead is acceptable, but for heavier programs like Firefox it can be several thousand times. Some suggestions of possible optimizations are given in section 4.4.1.

## 4.2. Related work

As discussed briefly above, there are many different approaches to dynamic taint tracing. Some require special hardware, or access to the source code. Since the purpose of this thesis project has been to develop a tool that can work on regular binary programs without access to additional information or special system support, the focus here will be on related work that takes a similar approach to dynamic taint tracing. While most publications focus on some specific application of taint analysis, this section focuses on the taint tracing techniques used and the precision of the taint information in different systems. It should be noted that it is often difficult to compare the different performance figures presented in papers side-by-side, since different authors use different performance evaluation methods. Most of the systems are also not publicly available for independent evaluation and performance measurements. Another problem, which further complicates comparisons of different taint tracing methods, is that many publications tends to focus

[16] Allocating new nodes and sorting them into linked lists, etc.

mainly on the specific application of taint analysis rather than the taint tracing method as such. Lastly, note that this section is not meant as an exhaustive listing of related work, but rather to highlight some different representative approaches to taint tracing.

Many systems store only one bit of taint information ("tainted" or "not tainted"). One such example is the tool Flayer [**12**], mentioned earlier. Flayer is implemented using Valgrind and stands out among other systems by storing taint on the bit-level, i.e. the bit are the smallest data unit that can carry taint. Most other systems use the byte as the smallest taint-carrying unit. No performance measurements are published for Flayer, but since the tool is based on a modified variant of the Valgrind tool Memcheck, an educated guess is that both tools have similar performance. The mean slowdown factor of Memcheck is about 20 times, according to [**24**]. Flayer is, similarly to the tool described in this thesis, meant as a more generic taint tracing tool, mostly aimed at security auditing. Other systems that store only one bit of taint information are TaintTrace [**18**] and LIFT [**17**], both implementing runtime exploit detection systems. These systems are designed with speed optimizations as one of their primary goals, having average slowdown factors of 5.5 and 3.6 times respectively, according to the papers.

The Dytan framework [**13**] is meant as a generic and extensible taint tracing system. It is implemented using the DBI framework Pin and uses bit fields to store taint labels. This approach allows a few different labels to be used, but the number of labels is usually restricted by the number of bits in the CPU-architecture's native word size (usually 32 or 64). One benefit of this approach is however that taking the union of two sets of taint labels can be implemented as a simple and extremely fast bitwise OR-operation. The time overhead for Dytan is about 30 times according to the authors.

Among systems that need more granular taint information one common approach is to store only one taint label per byte. This allows relatively fast taint tracing of every byte of input, but leads to information loss if a value is derived from multiple input bytes. One such example is the Vigilante system [**5**], which implements a form of intrusion detection and prevention system that can create exploit signatures for worm attacks. The authors also note indeed that their approach only works if critical parts of the exploit (e.g. the address to use in a return pointer overwrite) is present verbatim in the input, and that the system fails to properly handle cases where critical parts of the exploit is derived from multiple fields of the input. No performance data for the taint tracing engine itself is available for the Vigilante system. Another system that uses a similar approach is the TaintCheck system by Newsome *et al* [**4**]. It is based in an earlier version of Valgrind and records tainted input by storing taint structures, holding a copy of the tainted input bytes. Taint is then propagated by copying pointers to this taint structure to the appropriate locations in shadow memory. The paper reports an average slowdown factor of about 30 - 40 times.

Looking beyond systems implemented with DBI, there are a few other implementations related to this project. The BitBlaze binary analysis platform [**25**] can perform various kinds of dynamic and static analysis. Its dynamic analysis component TEMU uses whole-system emulation to perform taint analysis. TEMU is designed as a generic and flexible taint tracing system and provides an API to allow various types of taint analysis tools to be implemented. It can associate each tainted byte with a pointer to a user-defined data structure, which for instance would allow for an implementation of

multi-label taint propagation similar to this work. No performance figures are presented for TEMU itself.

Another work related to this one, in the sense that it also implements multi-label taint propagation, is the system for whitebox fuzzing described by Ganesh *et al* in [**11**]. Their approach however requires instrumentation of the *source code* and cannot work on the original program binaries. When having access to the source code the instrumentation can be made much more efficient, since taint can be propagated "directly" from one variable in memory to another without needing to trace taint through e.g. registers in all intermediary steps performed in the machine code. No performance figures are presented in the paper.

## 4.3. Multi-label taint propagation

The approximation of storing only one taint label is used in several systems, but not much work has been published on the validity of it. For some specific applications this approximation works adequately, but in the general case it will lead to information loss if a value is derived from multiple input bytes. To investigate the impact of this approximation a simple experiment was carried out by modifying the tool to record multi-label taint propagation statistics: Every time a byte tainted with multiple taint labels is used in an input operand to an instruction, a counter is incremented. The final counter value is then printed at program exit. The entire shadow memory is also iterated through at program exit and the number of multi-label bytes in memory is counted. This approach could in some cases lead to an overly pessimistic estimate of the information loss when doing single-label propagation: Consider the case of adding a constant to a tainted value. If e.g. a four-byte value is tainted with one label per byte according to [{1}, {2}, {3}, {4}] and a constant is added to this value, then according to the taint propagation semantics from section 2.2.1 the resulting taint will be [{1,2,3,4}, {1,2,3,4}, {1,2,3,4}, {1,2,3,4}]. If only a single label per byte would be used the result would instead be unchanged, i.e. [{1}, {2}, {3}, {4}]. If this value is used as a four-byte unit throughout the entire execution, the effective information loss would be small, even though the multi-label information is lost. (The information that the four-byte field *as a unit* depends only on labels 1, 2, 3, 4 is still retained.) If the value would instead be derived from two different tainted values, then important information would definitively be lost in the single-label approximation. The first case will from here on be referred to as a *self-mix* operation, i.e. sets of related taint labels are mixed with each other in the result. The second case, i.e. that taint labels from two unrelated byte ranges will be mixed with each other, will be referred to as a *full-mix* operation. To investigate if the first case is common, the linked list structure used to store taint labels was augmented to also store information of whether the byte had been derived using a full-mixing operation. In addition to all multi-label bytes, the occurrence of full-mixed bytes was also counted as described above.

Some general statistics on the total number of executed instrumented operations and taint propagating instructions were also collected by making additions to the analysis code added during instrumentation. Note that all operations referred to here are *primitive operations* in the VEX IR. Since complex x86 instructions are broken down to several primitive operations, the values presented here are larger than the number of executed native instructions.

Statistics were calculated for one execution each of the four programs used in the benchmarks, with each program's respective input file as a taint source. The results are presented below.

| | Total instrumented operations | Taint propagating operations |
|---|---|---|
| **Firefox** | 10,414,504,000 | 201,399,415 |
| **OpenOffice Writer** | 4,161,441,076 | 275,413,860 |
| **gcc** | 235,292,860 | 255,130 |
| **Lynx** | 149,038,401 | 1,893,031 |

**Table 3: Taint propagation statistics for four programs.**

| | Bytes in all input operands | | | Bytes in memory | | |
|---|---|---|---|---|---|---|
| | **Total** | **Multi-label** | **Full-mixed** | **Total** | **Multi-label** | **Full-mixed** |
| **Firefox** | 708,205,692 | 706,328,605 | 706,320,557 | 586,406 | 578,193 | 578,189 |
| **OpenOffice Writer** | 947,437,684 | 947,382,214 | 947,382,214 | 609,873 | 609,632 | 609,632 |
| **gcc** | 337,710 | 52,302 | 52,074 | 764 | 120 | 120 |
| **Lynx** | 3,210,473 | 1,024,565 | 1,005,581 | 9,703 | 1,351 | 1,282 |

**Table 4: Multi-label statistics for four programs.**

From Table 3 it can be seen that relatively few of the instrumented instructions actually propagated any taint. OpenOffice Writer has the highest fraction with about one taint propagating instructions in 15, while for gcc the fraction is about one in a thousand. Note that these statistics don't completely represent the entire picture, since e.g. floating point and SIMD operations are not instrumented.

Table 4 shows the multi-label taint propagation statistics. The first three columns show statistics for the entire data throughput of the program, i.e. every byte of every input operand of all executed instructions (except FP/SIMD ones). The next three columns show the corresponding statistics for all bytes in memory just before the program exited. It can immediately be seen that almost all multi-label bytes are in fact derived by a full-mix operation. For the larger GUI programs, almost all tainted bytes were also multi-label ones, while for the smaller programs considerably fewer bytes were tainted with multiple labels. For gcc, about 15% of all bytes operated on where multi-label ones. The figure was about the same for bytes in memory at program exit. For Lynx the corresponding values were about 30% and 15% respectively. One reason for this difference could be that, for the larger GUI-driven programs, a lot of processing happens in shared libraries for the graphics and GUI subsystems, which are unrelated to the "core logic" of the program. The complexity of a program also affects the fraction of multi-label bytes that are counted in input operands to instructions; the longer the chain of intermediate operations to arrive at a final result is, the smaller is the contribution of the original single-label bytes to the total sum.

In conclusion, these results show that the single-label approximation might work acceptably for smaller programs or for some specific applications, but that a general purpose taint tracing system must allow multi-label taint propagation to avoid loss of taint information.

## 4.4. Future work

Some suggestions on improvements of the prototype tool are given here.

### 4.4.1. Performance improvements

Performance optimizations can be of two categories, optimizations of the baseline performance or the taint propagation performance. One way to optimize the baseline performance is to implement more of the analysis code directly in the VEX IR, instead of relying on callouts to C functions. Since executing the "boilerplate" code to perform the actual call is relatively costly, as mentioned earlier,

large performance gains can be achieved by avoiding function calls. The Valgrind instrumentation mechanism is however not flexible enough to allow arbitrary analysis code to be implemented in IR; conditional statements can for example not be added to a superblock during instrumentation. One alternative would be to use highly efficient IR code to check if there actually is any taint to propagate, and only call the C function implementing taint propagation if this is the case. (Valgrind has a special mechanism to allow helper functions to be conditionally called from the IR.) Since actual taint propagating instructions are relatively rare, as can be seen from the results in section 4.3, this could considerably improve performance.

Apart from the baseline optimizations, there are multiple ways of improving the taint propagation performance over the current unoptimized prototype implementation. One obvious improvement would be to use trees with *O(log n)* insertion time instead of the current *O(n)* sorted linked lists. The reason for choosing linked lists in the first place is their lower memory consumption. In a binary tree each node needs to be at least 12 bytes; four bytes for the taint label plus one "left" and one "right" pointer. In a linked list only one "next" pointer is needed, thus reducing the memory consumption by 1/3. The choice of data structure is thus a tradeoff between execution time and memory use. Another improvement would be to optimize memory allocations. The current implementation uses the standard malloc algorithm, but a specialized algorithm for node allocation could probably improve performance considerably. Such an algorithm could for example exploit the fact that all list nodes are of the same size.

Finally, memory use could be decreased by optimizing the way taint sink events are stored. In the current implementation, a snapshot of the call stack is saved each time a "call" or "return" instruction is executed, and taint sink events store a pointer to the current stack trace at the time of the event. Since taint sink events that happen within the same function context share the same stack trace, some space is saved. The handling of stack traces could however be optimized further by only storing incremental changes of the call stack when a "call" or "return" instruction is executed. This would reduce the memory used for storing stack traces drastically by eliminating duplicate information.

### 4.4.2. Functional improvements

Since this implementation is just a prototype, there are numerous possible improvements and extensions that could be made to the tool. An assortment of possible functional improvements is listed here:

- Adding instrumentation of floating point and SIMD instructions to allow complete taint tracing of all programs.
- Implementation of the additional taint sink types in section 2.4.3.
- Support for other platforms than 32-bit x86 Linux. Currently, the tool is implemented specifically for one platform, but since Valgrind supports multiple platforms the tool could be extended to handle these as well. A limitation of Valgrind is that it lacks support for Windows. Since the greatest need for program analysis tools that does not need source code is likely to be in the Windows domain, porting the tool to Windows would improve its usefulness. This could be achieved by moving to another DBI framework, e.g. Pin or DynamoRIO. A tool implemented with one of these frameworks could also probably achieve better performance, since the overhead of Valgrind's code translation is avoided. The

drawback is that these frameworks don't support advanced features like system call wrappers, which means that much of Valgrind's advanced functionality would have to be re-implemented.

- Support for other taint sources, e.g. network sockets and command line parameters.
- Continuous writing of taint sink events to disk. In the current implementation, all taint sink events are stored in memory and written to the output file when the analyzed program exits. This means that the available amount of virtual memory limits the number of recordable taint sink events. A better approach would be to continuously store events to the output file as they occur.
- Real-time taint data. Instead of storing taint sink events to a file at the end of a taint tracing session, it would be possible to send taint sink events through e.g. a pipe or a socket to allow real-time analysis of taint sink events by an external program.
- More robust recording of stack traces. As mentioned above, the current implementation cannot accurately record stack traces in some cases. By taking various special cases, e.g. the behavior of the dynamic linker, into consideration better stack traces could be produced.

# 5. Conclusions

Dynamic taint tracing of binary programs is a powerful program analysis method that can be used for many applications within software security and other fields. While most current methods sacrifice precision in favor of performance, the prototype tool developed in this thesis project is capable of performing taint propagation with an arbitrary number of taint labels per byte. As expected, multi-label taint propagation results in considerably higher overhead than other approximate methods. Experiments performed on some real-world applications however show that considerable information is lost in the single-label-per-byte approximation. For some large GUI-driven programs almost every byte was derived in such a way that taint information would have been lost without multi-label propagation. For some smaller non-graphical programs the proportion was about 15-30%. While it is difficult to draw any definitive conclusions from such a limited number of measurements, a conservative estimate could be that at least 15-30% of all bytes used in most programs carry taint from multiple bytes of input. While the single-label approach works for some very specific applications, it is clear that a general purpose taint tracing engine must support multi-label taint propagation. Developing efficient and generic methods for this still remains an open research problem. Some proposed optimizations of the tool include replacing callouts to functions with inline analysis code, using more efficient data structures for storing taint and developing a specialized memory allocation algorithm for the tool.

The approach of using a complex data structure for each byte to store taint information results in high overheads, which makes analysis of very large graphical programs somewhat impractical. The benefit of this approach is however that the tool is easily extended to collect advanced per-byte statistics, such as those in section 4.3. This makes the tool well suited as an experimental platform for further research.

Another conclusion that can be drawn from the experiences of developing and using the prototype taint tracing tool is that constructing a truly generic such tool poses a considerable challenge. As briefly discussed in section 4.1, the performance measurements had to be performed with pretty conservative taint propagation and taint sink policies to avoid overtainting that would otherwise had resulted in unreasonably high overhead or the tool failing due to memory overuse. Due to the potentially extreme volumes of data that could be produced by taint tracing regular consumer software, a generic taint tracing tool would need to have very advanced features for letting its users specify precise constraints on taint propagation behavior and taint sinks. This is probably the reason why most existing implementations of dynamic taint tracing systems are specifically tailored for a certain application, and why attempts to develop generic taint tracing systems are so far fairly rare.

# Bibliography


[1] Microsoft Corporation. !exploitable Crash Analyzer - MSEC Debugger Extensions. [Online]. http://msecdbg.codeplex.com/

[2] V. Benjamin Livshits and Monica S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, 2005.

[3] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl*. Sebastopol, CA: O'Reilly and Associates, 1996.

[4] James Newsome and Dawn 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*, 2005.

[5] Manuel Costa et al., "Vigilante: end-to-end containment of internet worms," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[6] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[7] Nahid Shahmehri et al. (2012, February) An Advanced Approach for Modeling and Detecting Software Vulnerabilities (Draft). [Online]. http://www.ida.liu.se/~nahsh/sectest-draft.pdf

[8] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz, "Tupni: automatic reverse engineering of input formats," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[9] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[10] T. R. Leek, G. Z. Baker, R. E. Brown, M. A. Zhivich, and R. P Lippmann, "Coverage Maximization Using Dynamic Taint Tracing," Massachusetts Institute of Technology, Lincoln Laboratory, 2007.

[11] Vijay Ganesh, Tim Leek, and Martin Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[12] Will Drewry and Tavis Ormandy, "Flayer: exposing application internals," in *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.

[13] James Clause, Wanchun Li, and Alessandro Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007.

[14] "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture," 2011.

[15] Cristina Cifuentes, "Reverse Compilation Techniques," PhD Thesis 1994.

[16] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004.

[17] F. Qin et al., "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[18] W. Cheng, Q. Zhao,.B Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications*, 2006.

[19] Chi-Keung Luk et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[20] Derek Bruening, Timothy Garnett, and Saman Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *International Symposium on Code Generation and Optimization*, 2003.

[21] Nicholas Nethercote and Julian Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

[22] Valgrind web site. [Online]. http://valgrind.org

[23] Julian Seward and Nicholas Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the USENIX'05 Annual Technical Conference*, 2005.

[24] Nicholas Nethercote and Julian Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*, 2007.

[25] Dawn Song et al., "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Information Systems Security*.: Springer, 2008.

[26] Aleph One, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, vol. 7, no. 47, 1996.

[27] Eldad Eilam, *Reversing: Secrets of Reverse Engineering*.: Wiley, 2005.

# Appendix A: Benchmark results for the taint tracing tool.

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 477.49 s | 388412 kB |
| 2 | 490.57 s | 389624 kB |
| 3 | 492.14 s | 371152 kB |
| 4 | 481.01 s | 380388 kB |
| 5 | 465.19 s | 369104 kB |
| Average | 481.28 s | 379736 kB |

**Table 5: Firefox, taint tracing with no taint.**

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 6033.87 s | 389396 kB |
| 2 | 494.83 s | 375060 kB |
| 3 | 10881.0 s | 631992 kB |
| 4 | 527.28 s | 382228 kB |
| 5 | 536.31 s | 383252 kB |
| Average | 3694.67 s | 432386 kB |

**Table 6: Firefox, taint tracing with taint.**

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 261.72 s | 419468 kB |
| 2 | 256.68 s | 418388 kB |
| 3 | 256.33 s | 419468 kB |
| 4 | 258.34 s | 419332 kB |
| 5 | 263.45 s | 418348 kB |
| Average | 259.30 s | 419001 kB |

**Table 7: OpenOffice Writer, taint tracing with no taint.**

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 762.05 s | 503428 kB |
| 2 | 918.22 s | 512652 kB |
| 3 | 945.39 s | 515704 kB |
| 4 | 266.98 s | 403328 kB |
| 5 | 939.74 s | 516624 kB |
| Average | 766.48 s | 490347 kB |

**Table 8: OpenOffice Writer, taint tracing with taint.**

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 19.84 s | 94756 kB |
| 2 | 19.67 s | 94756 kB |
| 3 | 19.87 s | 94756 kB |
| 4 | 19.72 s | 94756 kB |
| 5 | 19.84 s | 94756 kB |
| 6 | 19.78 s | 94756 kB |
| 7 | 19.66 s | 94756 kB |
| 8 | 19.62 s | 94756 kB |
| 9 | 19.65 s | 94756 kB |
| 10 | 19.67 s | 94756 kB |
| Average | 19.73 s | 94756 kB |

**Table 9: gcc, taint tracing with no taint.**

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 19.91 s | 94756 kB |
| 2 | 19.88 s | 94756 kB |
| 3 | 20.04 s | 94756 kB |
| 4 | 19.87 s | 94756 kB |
| 5 | 19.81 s | 94756 kB |
| 6 | 19.85 s | 94756 kB |
| 7 | 19.86 s | 94756 kB |
| 8 | 19.9 s | 94756 kB |
| 9 | 19.93 s | 94756 kB |
| 10 | 20.01 s | 94756 kB |
| Average | 19.91 s | 94756 kB |

**Table 10: gcc, taint tracing with taint.**

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 9.42 s | 54508 kB |
| 2 | 9.41 s | 54508 kB |
| 3 | 9.46 s | 54508 kB |
| 4 | 9.44 s | 54508 kB |
| 5 | 9.85 s | 54508 kB |
| 6 | 9.43 s | 54508 kB |
| 7 | 9.46 s | 54508 kB |
| 8 | 9.45 s | 54508 kB |
| 9 | 9.41 s | 54508 kB |
| 10 | 9.46 s | 54508 kB |
| Average | 9.48 s | 54508 kB |

**Table 11: Lynx, taint tracing with no taint.**

| Test | Execution time | Memory use |
|---|---|---|
| 1 | 10.03 s | 54196 kB |
| 2 | 10.08 s | 54196 kB |
| 3 | 10.07 s | 54196 kB |
| 4 | 10.11 s | 54196 kB |
| 5 | 10.06 s | 54196 kB |
| 6 | 10.15 s | 54196 kB |
| 7 | 10.06 s | 54196 kB |
| 8 | 10.12 s | 54196 kB |
| 9 | 10.08 s | 54196 kB |
| 10 | 10.07 s | 54196 kB |
| Average | 10.083 s | 54196.0 kB |

**Table 12: Lynx, taint tracing with taint.**