Documented Data Dispersal

A Thesis presented

by

Frederic H. Behr, Jr.

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 8, 2003

Thesis advisor

Professor Michael D. Smith

Author

Frederic H. Behr, Jr.

Documented Data Dispersal

Abstract

We define the *Documented Data Dispersal (DDD)* problem as determining, given inputs, outputs, and an application connecting them, whether or not a certain input was used to produce a certain output. There exist many possible ways to address this problem. All of the solutions can be characterized by the level of program analysis, instrumentation, and data analysis that they perform. These parameters determine the tradeoffs between risk and performance that a particular solution makes.

We first evaluate similarity metrics as a method of addressing the *DDD* problem. This method cannot detect all potential data dispersal but has characteristics that make it ideally suited for some domains, such as plagiarism detectors. Next, we evaluate binary rewriting by instrumenting common functions that perform data movement, thereby tracking memory buffers that contain data from a particular input file. This method is very fast; it introduces almost no overhead. Unfortunately, it also misses significant amounts of data movement. Finally, we evaluate a binary interpreter as a means to dynamically identify instrumentation points. This method incurs significantly more overhead than the first two but promises to mitigate more risk as well

Analyzing the parameters described above is a powerful tool that can be used to determine the risk/performance tradeoff that a method makes. Using this tool, we argue that future development of methods to address the *DDD* problem should start from a position of no risk and terrible performance and explore the performance enhancements that can be made while minimizing the risk. This bottom-up approach appears to have promise.

Acknowledgments

Thank you to my advisor, Mike Smith. His guidance and help were invaluable. Thanks to Mike, Dave, Julie, Omri, Mom, and Dad for providing valuable comments. This thesis is better because of you. To all my friends and family for their encouragement. Finally, to Kristina, for keeping me sane.

Contents

Chapter 1

Introduction

It is very difficult for typical users to know exactly what a piece of software installed on their computer is going to do. They must trust that the guarantees made by the author of a program are accurate.[1] Unfortunately, in many situations it is not wise for a user to trust those guarantees. In some instances developers may intentionally misrepresent their software. Additionally, even the best-intentioned software vendor cannot promise that its software will do no harm. No one can be sure that any reasonably sized software package is bug-free. Anytime a user runs software that is untrusted, the program may silently perform some undesired action, either due to a malicious developer or buggy code.

In this thesis we focus on understanding methods for detecting the unwanted dispersal of data. This is one of the most important ways a malicious program can misbehave, because, in many situations, users more highly value the data files on a computer than any other part of the system. While important data files are often protected by a security policy that should govern their use, on a normal system malicious programs are capable of silently violating these policies. Our goal is to provide a mechanism to break this silence—to monitor and detect undesired data dispersal.

We believe that this is an important silence to break. Many vendors do not have their consumers' best interests in mind. The internet has exacerbated this problem by enabling the proliferation of easily downloadable software. It is difficult for an injured user to hold culpable faceless vendors that freely distributes their software on the web. Recent revelations of "spy-ware" (programs that transparently track users' actions) incorporated into popular internet software highlight this problem [15]. One can imagine more malicious

[1]Though, even if *they* are the authors, an evil compiler could cause surprises.[19]

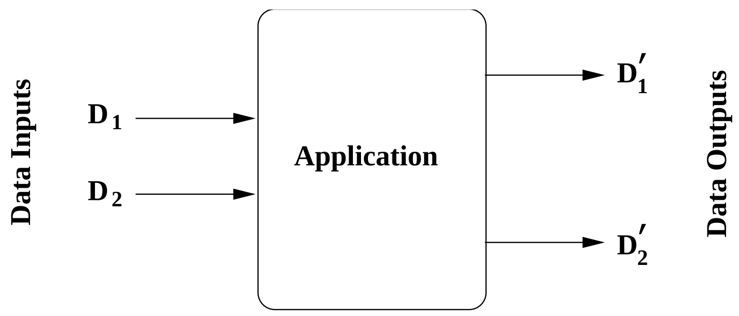

Figure 1.1: Our goal is to correlate data between inputs and outputs. Did data from D_1 get written to any D'? How is D'_1 related to the Ds?

programs as well. For instance, a peer-to-peer application could search the user's hard drive for financial records and send them out over the network. In this situation, the users of a program need assurances that an untrusted program is not maliciously using their data.

In a similar situation, system owners could want assurances that users are not maliciously transferring data. In circumstances where people are trusted with confidential information, the owners of the data may want security measures in place over that information. For instance, many companies distribute sensitive information about upcoming product launches to employees. A system that notified them if this information was emailed to a third party or copied to removable storage would help deter leaks and enable appropriate action. A recent PriceWaterhouseCoopers report suggests the magnitude of the problem such a system would address, stating that between \$53 and \$59 billion in damages was reported due to proprietary information loss among 138 survey respondents in 2001 [14]. Clearly this is an important problem to address.

We explore a range of potential solutions to this problem. First we discuss using an existing algorithm to estimate the similarity of the inputs to the outputs of a program. Next we use a runtime binary rewriting tool to insert bookkeeping code into data movement library functions such as `memmove`. Finally we explore using a binary interpreter to dynamically inspect programs for data dispersal. We discuss the accuracy and the performance overhead associated with each of these methods. This analysis allows users to determine the appropriate method for their particular situation.

1.1 Problem Statement

We say that data flows into or out of an application through *channels*. We define the problem of ascertaining whether an application created data on an output channel using data from an input channel as the *Documented Data Dispersal* (*DDD*) problem. A solution to the *DDD* problem can be used to address the situations described above.

Channels include such mechanisms as files, sockets, inter-application memory buffers like the Microsoft Windows clipboard, the character stream coming from a user's keyboard, and the output of an application displayed on the user's computer screen. For ease of discussion, we will use files as the canonical example of input and output data channels, though the techniques and findings of this thesis are applicable to any type of data channel.

Figure 1.1 illustrates the *DDD* problem for the simple case of two input and two output channels. A method that solves the *DDD* problem would be able to answer the following types of yes/no questions:

- Did the application use data in file D_1 in generating file D'_1?
- Did the application use data in file D_1 in generating file D'_2?

In general, an application may have many input and output channels, and the number of each does not have to be the same. Though we restrict our analysis to one input file, the methods presented here generalize to the common case where programs have many input channels. In addition, we consider data generated by the program without the use of any external data as simply another input channel.

We exclude from our investigations input data that is used to direct the control of an application. For example, when a script is loaded and interpreted, some of the output of the interpreter is determined by the commands in the input script. We do not consider an output file as generated from data contained in an input script, even if the commands in that script directed the interpreter to produce the output file. Technically, we consider scripts to be specializations of the interpreter program, and given this specialization, we can solve the *DDD* problem on the resulting program's inputs and outputs.

At first glance, the *DDD* problem may seem trivial. You could compare the two files of interest (e.g., D_1 and D'_1) to see if their contents are identical. Going beyond checks for exact matches, search engines like Google and AltaVista have commercialized technology that help identify the "similarity" between two files [3]. Unfortunately, many commonly used applications, such as file compression and encryption tools, manipulate input data so that there appears to be no similarity between the input and output files. The result is that no general algorithm exists for solving the *DDD* problem given only an input and output file.

The *DDD* problem is a subset of a traditional security problem called "non-interference". Non-interference deals with preventing data flow by all means. This includes the channels that we consider, as well as scripts, the existence or non-existence of temporary files, and irregular timings—anything that could be used to leak information from one process to another [13]. These other means are sometime called "covert channels". We have scaled down this problem while maintaining its practical significance. Furthermore, the *DDD* problem focuses on detecting and documenting data dispersal. The related problem of preventing unwanted data dispersal is also important, though not the topic at hand.

This thesis is concerned with solutions to the *DDD* problem given an input file, an output file, and an application connecting the two files. Some work exists that addresses this problem via static analysis of the application [16]. We are interested in solutions that require only an application binary and not the application's source code. Furthermore, we are interested in runtime solutions that do not require instrumentation or recompilation of the application's image on disk. All such approaches run counter to commercial realities: application vendors do not typically distribute source code (especially malicious developers), and end users are not interested in modifying the binaries that they have downloaded. This thesis describes implementations of our methods for Microsoft Windows running on an Intel x86 processor. Though the implementation has platform dependencies, the techniques we present are not platform specific.

1.2 The Risk/Performance Tradeoff

Users may need and want considerably different levels of security for different situations. No one security policy is appropriate for all users and all environments. Because of this, all users will not be willing to go to the same lengths to gain the security that they need. A typical home user who needs to protect digital bank statements may not be willing to sacrifice as much as a large corporation looking to protect valuable intellectual property.

We discuss both the risk and the performance associated with methods to solve the *DDD* problem. In our work we categorize the risk associated with a method via a risk matrix. This matrix clearly outlines the types of data dispersal on which the method does well, and those on which it does poorly. In our work we measure performance as the computational overhead incurred in a detection system.

The methods we evaluate to answer the *DDD* problem checkpoint data at a range of granularities, which translates to a range of both performance and accuracy. At the highest level we treat the program as a black box. We can observe the input channels and the output channels, but we have no way of observing what is done in the application. In this situation, we simply perform a comparison of the files read by a program to those written by it. Clearly, this does not affect the performance of the application as all of the analysis is done offline. However, this method is also unable to document all types of data movement. At a lower level we examine the paths taken by the data inside the program. We attempt to track the memory buffers that contain data from some input file. Our two runtime methods differ in the granularity at which they track these buffers. The methods can gain accuracy by using a finer granularity, but at the cost of sacrificing performance. The runtime binary rewriting method tracks buffers at the level of library calls while binary interpretation can examine every function call or more.

There is no such thing as perfect security in a system. If users store data somewhere in their system, there is a chance that it will be compromised. None of the methods that we evaluate will solve the *DDD* problem outright. Rather they each provide us with insights into the nature of the risk/performance tradeoffs that can be made and into the shape of an eventual practical solution to the problem.

1.3 Outline

Chapter 2 presents in detail the three methods that we analyze. Chapter 3 provides an analysis of the risk/performance tradeoffs associated with these methods. We apply our results from the previous chapters in a discussion of future directions in Chapter 4. We then describe work related to the *DDD* problem and our methods in Chapter 5.

Chapter 2

The Methods

Methods to address the *DDD* problem all have the same behavior when viewed from a high level: they perform *program analysis* on an application in order to insert *instrumentation* that does *data analysis*. We call these three characteristics the *DDD solution parameters*. The methods differ in their implementations of the solution parameters. The analysis of a program can be done at different levels; it can look at machine instructions, blocks of code, function calls, or any other unit of code. The methods can insert instrumentation at a similar range of levels. The sophistication of the data analysis can also vary. These parameters define the characteristics of a method—how much risk it mitigates and how much overhead it incurs. This chapter describes the methods that we have explored and their solution parameters.

2.1 Similarity Metrics

The first method we evaluate to address the *DDD* problem is a static similarity method. This method gives us a picture of the types of transformations that an application can perform on the input data while still allowing us to estimate the origin of output data with an offline analysis. There are many different types of similarity metric algorithms. For example, there are definitions of the "distance" between one sequence of characters and another. The Hamming Distance between two strings is defined as the number of indices into the strings in which they differ. The Levenshtein Edit Distance is defined as the minimum number of edits (insertions, deletions, or substitutions) that it takes to transform one string into another [5].

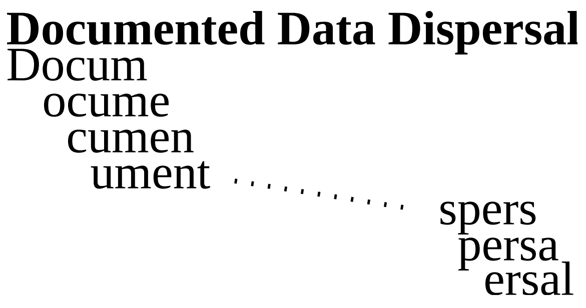

Figure 2.1: Shingles of five bytes for the phrase "Documented Data Dispersal".

The similarity metric that we have chosen was first described by Broder. It allows us to identify documents that are "roughly the same" [3]. This metric uses the notion of overlapping "shingles" in a document as the basis of its computation. A single shingle is a fixed-size substring in the document.[1] Because we look at all of the substrings of this given size during the algorithm; these substrings overlap like shingles. If we take two documents and put their respective shingles into two different sets, then the intersection of the sets will tell us how many shingles they have in common. If the shingles of two strings are sized appropriately, and a large percentage of them are in common then the two strings must be very similar.

The appropriate size for a shingle depends on the granularity of the data that is significant. For instance, if this algorithm is run on text and the shingle size is set to be a word, then it will compute how many words the two documents have in common. If each shingle is five words long, you will compute how many phrases the two have in common. Setting this parameter too small will give a false impression of document similarity; it is not really informative that two documents have a large number of letters in common. Setting the parameter too large risks overlooking two similar files; if one simply inserts a random byte after every hundredth byte and the shingle size is two hundred bytes, the algorithm will not detect any similarity.

Broder described an algorithm to efficiently estimate the number of shingles that the two documents in question share [3]. This algorithm takes time linear in the size of each document for a preprocess sketch. After sketching each document it takes a small constant time to compute the similarity of any pair of documents.

[1]The size here can refer to any number of *tokens*, where a token can be a byte, a character, a word, or any other unit of data.

To use the similarity metric to address the *DDD* problem we precompute the sketch of the file that we are concerned about, and then compute the sketch of output files from an application. We can then compute the similarity of two files. If the similarity is above a certain threshold, then we know that data has been dispersed. The particular threshold that indicates data dispersal should be determined by the parameters to the similarity metric. The user must have some way to determine the input and output files in order for this method to be used. This could be done by binary rewriting, as discussed below, or by some other method.

Similarity metrics have interesting *DDD* solution parameters. They do all of their work offline, meaning they perform no program analysis. They may instrument the application only to find the inputs and outputs, or they may perform no instrumentation at all if they employ another method to estimate the inputs and outputs. To make up for the minimal analysis and instrumentation, similarity metrics must employ a sophisticated data analysis algorithm—in our test case the Broder similarity metric.

2.2 Dirty Buffer Tracking

The remaining methods we explore both attempt to address the *DDD* problem by tracking dirty buffers. At a high level, a buffer is *dirty* if data from the input file was read directly into it, or if data from an already dirty buffer was read into it. We use a binary interval tree to record dirty buffers. Each node in this tree has a minimum and a maximum address. No nodes overlap. When we instrument data movement we check if the source buffer is dirty. If so, we insert the destination buffer into the tree, in effect marking it dirty. Otherwise we clean the destination buffer by removing it from the interval tree if it is already present. Thus, the tree records the address ranges that we believe to be dirty. When the program writes to a file, we document data dispersal if the data written is present in any of the data ranges in our tree.

This implementation tracks buffers that are dirty with data from a single input file. In a more general solution we would keep track of equivalence classes of buffers—those that are clean, those that are dirty from the first input, those that are dirty from the second input, etc. Note that it is possible for a single buffer to be dirty with data from multiple inputs. For example, if a buffer is the exclusive-or of two other buffers, then both of the source buffers could dirty the destination.

Function Name	Description
`strcpy`, `lstrcpy`, `lstrcpyW`, ...	String copy functions
`memcpy`, `wmemcpy`, `MemoryCopy`, ...	Memory block movement functions
`memmove`, `wmemmove`, `MemoryMove`, ...	Memory block movement functions, allowing overlapping blocks
`ReadFile`	Reads data from a file to a buffer
`GetClipboardData`, `SetClipboardData`	Gets and sets the text in the copy/paste buffer

Figure 2.2: A partial list of the data movement functions that we have instrumented.

This sort of buffer tracking is only one possible choice for the data analysis solution parameter. The similarity metric shows another, and one could imagine more—for instance, marking entire memory pages dirty and clean. We believe the buffer range approach is a sensible one. It allows us to follow data at a very fine granularity, yet does not entail an inordinate amount of record keeping.

2.3 Binary Rewriting

The second method to address the *DDD* problem employs a binary rewriting tool that allows us to dynamically monitor data dispersal during runtime. The tool is produced by Liquid Machines and lets us insert our own code in place of library functions [18]. Using this tool, we add data analysis code to the entry points of functions such as `memmove` and `strcpy` that transfer data from one area of memory to another. This allows us to follow buffers that contain data from the input file that we are concerned about.

This tool can hook any function of which it is able to find the entry point. This means that any function in a dynamically loaded library (DLL) can be hooked. Arbitrary functions in the application can only be hooked if their address is known. This makes it difficult to hook use defined functions. We need to hook functions that will move data from one buffer to another. Fortunately, many of the standard data movement functions are included via system DLLs. Figure 2.2 presents a partial list of the functions that we have hooked.

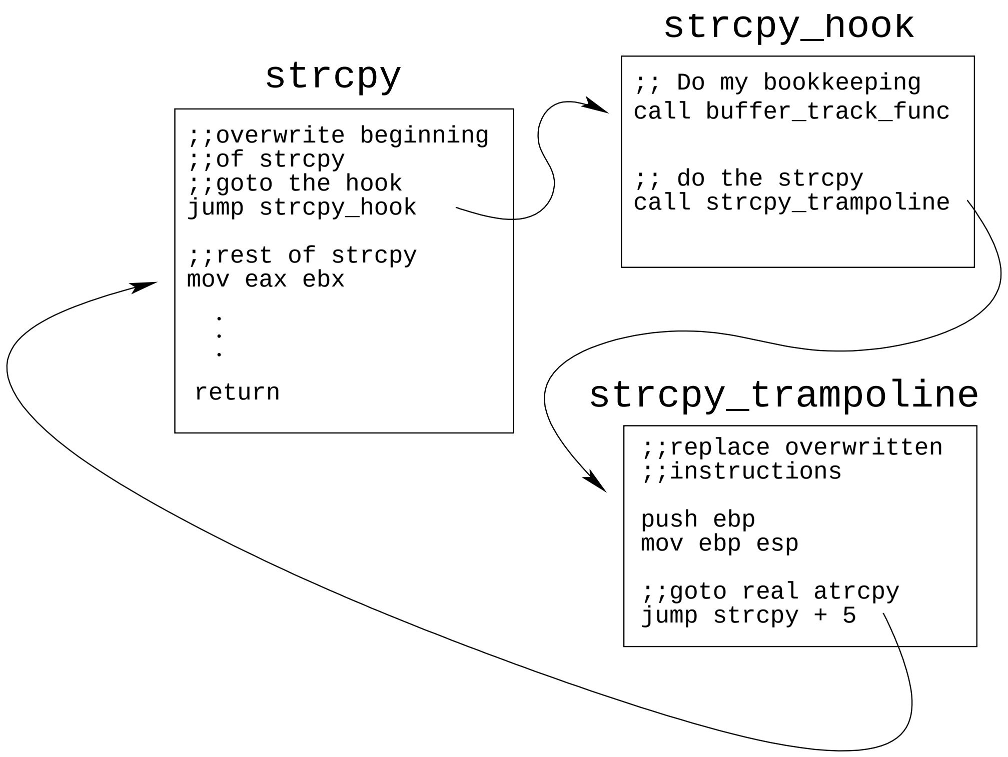

Figure 2.3: Mechanism for hooking function calls. The first instruction of the function is overwritten with a jump to our code, then we have a trampoline that runs the overwritten instructions and jumps to the rest of the hooked function.

The binary rewriting tool does its work as the binary loads and as new DLL's get loaded. It overwrites the first few bytes of the target function with a jump instruction to our code. This is called "hooking" the function. It also creates a "trampoline" stub that executes the overwritten instructions and then jumps to the rest of the target function. Our code can make a call to the trampoline stub, which in effect acts just like a call to the function that we have hooked. This allows us to call the target function from our code. Figure 2.3 shows an example of the `strcpy` function being hooked. The original function is `strcpy`, the code that we execute in its place is called `strcpy_hook`. Notice `strcpy_hook` calls `strcpy` using the trampoline.

The hooks that we add all follow the same general pattern. They begin with the data analysis code described above. The hooks then use the actual function that the application called. Let us use `strcpy` as an example, illustrated in Figure 2.3. This function is declared `char* strcpy(char* dest, char* src)`. In our hook we first check to see if `src` is a dirty buffer . If so, we mark the range from `dest` to `dest + strlen(src)` as dirty. If `src` is *clean*, then we mark the range from `dest` to `dest + strlen(src)` as clean. In either case, we then call the real `strcpy` using the trampoline, and return the result of that function call.

This method allows us to keep track of the buffers that have been dirtied by library functions that move data from buffer to buffer. We use this to address the *DDD* problem by dirtying any buffer that gets data from the input file read into it. We then follow the dirty buffer as the application is running as described above. At every call to write data to a file we check to see if a dirty buffer is being written. If so we have detected data dispersal. This is partial solution to the *DDD* problem, data movement that takes place outside of hooked functions will not be detected.

This method does significantly more analysis and instrumentation than the similarity metric, but the bookkeeping is simpler. Though there is now analysis, it is done offline. The analysis consists of determining which functions to hook. There is runtime instrumentation using this method. The instrumentation code is inserted at the function level. The bookkeeping is fairly straightforward in comparison to the similarity metric, simply keeping track of buffer ranges.

2.4 Binary Interpretation

As we will see, binary rewriting is not a perfect solution to the *DDD* problem. It will be evident that a finer grain of program analysis is needed to properly address the problem—runtime program analysis.

We use a binary interpreter, which allows us to examine every instruction before it executes, to give us this finer grain of control over our instrumentation. We have implemented a binary interpreter and used it to experiment with the viability of this technique for solving the *DDD* problem. There are many instrumentation policies that one could use to address the *DDD* problem, each with a risk/performance tradeoff associated with it. We have explored one such policy: inserting data analysis code at the beginning of each function call.

A binary interpreter works much like a traditional interpreter; it examines and executes code statement by statement, though, here each “statement” is simply a machine instruction. Traditionally, this technique has been used to run a binary compiled for one

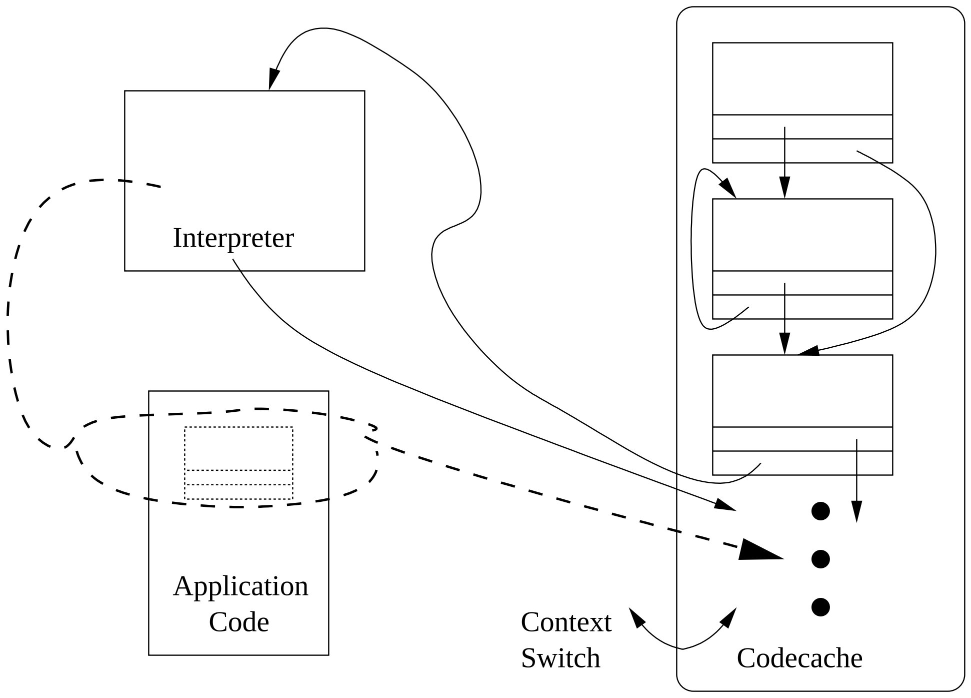

Figure 2.4: The binary interpretation system. The interpreter moves blocks of code from the application binary in to the code cache on at a time. The application then runs from the code cache.

ISA on top of another [4][9][11]. We use the interpreter to insert and run instrumentation code dynamically.

Our interpreter uses a caching technique to decrease overhead. The interpreter examines a block of code until it reaches an instruction with multiple possible exit points, such as a conditional branch or an indirect jump. It then puts this entire block into a cache and executes the code in the cache. At the end of the block there are two possibilities. If the next instruction happens to be at the beginning of a block that is already in the cache, then we jump there directly. Otherwise, we reenter the interpreter and continue the interpret-execute loop. In order to reenter the interpreter the processor must perform a costly context switch, thus we try to avoid this path.

The code cache dramatically improves performance over pure interpretation, but note that the interpreter still touches every instruction before it runs. This means that we could instrument every instruction if we needed to. This is not necessary to solve the *DDD* problem; we hope to use the interpreter framework to instrument the code at a much coarser granularity. We explore instrumenting the binary at the function call level.

We have not yet finished the implementation of using the interpreter to perform program analysis. We present a high level view of one possible approach here.

There are two things that we must determine via the interpretation program analysis: the first is where to insert instrumentation, and the second is what instrumentation to insert. The first is easy, we insert instrumentation at call sites, which are readily identified by the call instruction. The second, determining what instrumentation to insert, is more difficult. If a particular call site is not associated with a function that moves data, then we do not insert instrumentation. Otherwise we need to determine what parameters should be given to the buffer tracking data analysis code; what parameters are the source and destination buffers of the data movement. This is not trivial. Initially, we limit our analysis to observing data movement functions that obtain the source and destination buffers via function arguments. Even with this restriction, though, it takes some work to find these arguments. We assume that the memory range between the base pointer and the stack pointer is the current stack frame, and that all memory words on the current stack frame could be arguments to this function. This is a conservative estimate of the arguments to the function.

With an estimate of the arguments of the function in hand, we must determine which way, if any, that data flowed between buffers pointed to by these parameters. We do this using virtual memory support. We write protect each memory page to which the arguments point. This means that we will get a memory fault if the function tries to write to a buffer pointed to by one of the arguments. When this happens we can analyze the code surrounding the faulting instruction to determine the direction of the data flow. We note this direction in a table, unprotect the page so the write can occur, and return from the fault handler.

When we reach the return site of the function we can finalize the analysis. At this point we have determined the direction of the data movement in the function, so we can insert the instrumentation code and unprotect the memory pages. The instrumentation code that is inserted is a call to the same buffer tracking function that we use in the binary rewriting method. While there are many implementation details for this method that we have left unaddressed here, they are unimportant for the analysis of this method.

This method does substantially more sophisticated program analysis than the binary rewriting method. This enables it to identify instrumentation points that were impos-

	***DDD* Solution Parameters**		
Method	**Program Analysis**	**Instrumentation**	**Data Analysis**
Similarity Metric	None	Very little, if any	Sophisticated algorithm
Binary Rewriting	Offline determination of functions to hook	Data moving library function calls	Track dirty buffer ranges
Binary Interpretation	Runtime interpretation/ dataflow direction detection	All data moving function calls	Track dirty buffer ranges

Figure 2.5: An overview of the solution parameters for the the three evaluated methods.

sible to instrument before. This is the only difference between the methods though. The instrumentation is still inserted at the function level, and the data analysis still involves tracking dirty buffer ranges using an interval tree.

Chapter 3

Analysis

We evaluate methods to address the *DDD* problem along two axis: the risk they mitigate and the performance overhead that they incur. Both these measurements may be equally as important to a user trying to decide which method is appropriate. Users must make a tradeoff—how much risk do they need to mitigate as opposed to how much performance overhead they are willing to incur. In this chapter we present our experiments and analysis of this tradeoff for the three methods outlined above.

3.1 Method of Risk Analysis

It is an oversimplification to say that the higher the accuracy of the methods the lower the risk. We find that each of the three methods is suitable for a certain domain of problems. Users should judge the appropriateness of a method for their particular needs.

In order to judge this appropriateness, it is important for a user to have an understanding of how a method can succeed, and how it can fail. The *DDD* problem has two possible answers: data was dispersed, or it was not. Similarly, our methods will return one of the same two answers. This gives us four possible situations:

- **True Positive**: Data actually was dispersed, and our method recognized this fact.
- **True Negative**: Data was not dispersed, and our method recognized this fact.
- **False Negative**: Data was dispersed, but our method failed to recognize it.
- **False Positive**: Data was not dispersed, but our method claimed that some was.

		Expected Answer	
		Yes	No
Given Answer	Yes	True Positive	False Positive
	No	False Negative	True Negative

Figure 3.1: Risk matrix for the four possible situations given an answer to the *DDD* problem.

Both of the True cases are successes for our methods. The True Positive is exactly the situation that we are concerned about. The True Negative case is also a correct answer, but slightly less interesting. Users would probably prefer a True Negative, since it indicates no attempted dispersal. Both of the False cases are failures. A False Negative is very bad, as it indicates no data dispersal when there was, in fact, data dispersal. This may even be worse than not using these methods at all, since it can give users false confidence that their data is safe. A False Positive is also a problem, though not quite as insidious. This result can lead to a situation where a user does not believe a True Positive—akin the the problem of "crying wolf."

We determine the characteristics of applications that cause these situations for our three methods. Each method is different, so the interesting test cases differ across the methods. We first describe the common framework for the tests below. We then describe our risk analysis for each of the methods. This will allow a user to determine if a particular method is suitable for their needs based on the particular characteristics of that method.

3.1.1 Test Cases

Applications use data in a broad variety of ways. For instance, a text editor simply displays data to be read and manipulated by a user, a graphics editor processes data before displaying it to a user, and a compression utility transforms data but never shows it to the user. The best methods for addressing the *DDD* problem should be able to perform well regardless of how an application disperses the data in question.

We use a suite of applications to test the accuracy of our methods of documenting data dispersal. We chose our test applications to reflect the variety of types of data that applications use. The test applications process textual data, binary data, and a mixture of the two. We assume that applications process these different types of data in different ways. We claim to use a representative sample of applications in this respect. It is not possible to choose a representative for every conceivable type of application, but we believe we have picked a set of test applications that covers a broad range of the sample space.

The first applications that we use are simple text editors: Microsoft's Notepad and an open source version called PopPad. These two applications display and manipulate exactly the data that is on the disk. They do not use headers, footers, or other metadata. Notepad is a useful test case because it ships with every copy of Microsoft Windows, making it extremely widespread. PopPad is useful because we can examine the source code and thus support the analysis of our results. Furthermore, while both of these programs perform the same basic function, we have found that they use different implementation methods to do so.

We use Microsoft Word as another test application. Word also presents textual data to the user for display and editing. However, Word's document format contains a lot of metadata that the user does not see. This complicates the *DDD* problem for two reasons. First, Word uses different data in the file in different ways. Thus, a runtime method will have a more complicated data flow to monitor. More importantly, it is unclear if we want metadata to affect the results of our tests in the same way that regular user data does. It is possible that metadata could skew the results in both directions. For instance, two Word files with different user text could have identical metadata, making them seem more similar than they might otherwise. Furthermore, two Word files with the same user text may have very different metadata, thus skewing the results in the opposite direction.

We use the image editing program Microsoft Paint to test a third class of applications. Paint can manipulate and display a number of image formats. We run our tests using the bitmap image format. This format does not employ any compression; it simply stores the color value for each pixel. The application displays and lets the user edit exactly the data that is on the disk. In a sense, this is the image equivalent to Notepad or PopPad. Paint is interesting as a test case because it moves away from textual data. This allows us to explore the efficacy of our methods for applications that do not simply manipulate strings of text.

Lastly, we test our methods using the WinZip compression utility. This application differs from the others in that it is not an editor of any kind. This means that the application behavior will most likely be different. We assume that editors will all act the same way from a high level perspective; they store data in a buffer and perform edits on the buffer. The model for WinZip seems fundamentally different. It is using the input file as the argument to a compression function and outputting the result. This also means that the data file that it writes out is significantly different from the input. That is, in the previous applications data that moves from one file to another will most likely look similar to a user. In this application this is not the case, data that moves from an input file to an output file will look superficially very different.

When we use each of these applications with controlled inputs, we know what the application is supposed to do. Therefore, we have a good idea what the correct answer to the *DDD* problem is in each case. For instance, if we open a file in Notepad and save it with another name, we know that data has been dispersed. Similarly, if we open a document in Word, but leave it in the background and save an unrelated file in the foreground data has not been dispersed. It is possible that the applications produce temporary files that we are not initially aware of, but we do know for certain the major input and output files. Also, depending on our test we can be certain that data either did or did not disperse from the input file to the output file. This allows us to judge the effectiveness of our methods against the correct answer.

3.2 Method of Performance Analysis

Performance is the second important evaluation axis for these methods. We analyze the runtime performance of the methods in two ways. We first provide the overhead associated with the methods by measuring the difference in wall clock execution time of the original application to the execution time for applications running under the three methods. This will provide a sense of the overhead that a user will feel when using our implementations of the methods. We also determine the expected overhead via an analytic model of the methods. This enables a judgment about the expected overhead of the methods after the implementations have been more fully engineered and optimized.

Our measurements were done using Notepad and WinGrep, a graphical implementation of `grep`. The performance overhead measured in Notepad should be indicative of the overhead seen in an interactive setting. The overhead associated with WinGrep is meant to give an indication of the overhead in a non-interactive setting—it does a considerable amount of processing, and also copies buffers to store results, thus running any instrumentation code that was inserted.

In the Notepad test we open a one page test file, delete a sentence, and save the test file. We repeat this sequence five times. For the WinGrep test we search though a 1 GB file for a term that appears roughly 200,000 times. We repeated each test at least ten times. The large number of repetitions is especially important for the Notepad test as it involves significant user interaction, which has high variability. The median time for the Notepad test without using any data dispersal detection was 31 seconds. The high was 35 seconds and the low was 27. This corresponds to roughly 6 seconds for each open-delete-save sequence. The median WinGrep time was 321 seconds. The high was 335 seconds and the low was 312. See Figures 3.5 and 3.6.

3.3 Similarity Metrics

We found that the actual numerical data for the similarity of different documents does not interest us because it was very dependent on the particular parameters used. Our discussion focuses on the general trends and what they mean for the *DDD* problem. We show that this type of algorithm is inherently limited when used to address the question at hand. No static method can overcome these limitations and provide the full answer that we seek.

Though this approach is limited it also has benefits. It is simple to implement as it does no program analysis and little instrumentation. The offline characteristic could be important in some situations. First, it means that there is no runtime overhead. Also, this means that the method can be performed without access to the application binary—which could prove useful. Furthermore, though the method cannot address all of the questions that we want answered, it provides accurate information about a subset of them. The similarity metric can detect when data is simply copied from one place to another. This technique is also valuable because it allows us to get a high level feel for what a program is doing with the data that is input. The very fact that a similarity metric does not perform well in some experiment provides information about the processes of the application being tested.

As expected, the algorithm reliably detected when plaintext files were read into Notepad and written back out without modification. The files had not changed at all and were found to be 100% similar. Furthermore, when we opened a file in Notepad and replaced roughly half of the document with garbage, the similarity metric found that the documents were roughly 50% the same. In this test we replaced large chunks of the data. The shingle value easily fit within these chunks. When we replaced about the same amount of data, but in much smaller, more frequent chunks, the similarity score fell drastically. The actual score is very dependent on the size of the chunks versus the size of the shingles, but the pattern we see is that the closer together the modifications, the lower the similarity score. Likewise, when we replaced the entire document with garbage, we found 0% similarity. These results are all expected based on the characteristics of the similarity metric.

More interesting were the results of the tests with Microsoft Word. Word has about 20 KB of headers, even in a blank document. Also, these headers are not all identical. Two blank documents created one after another share about 90% of their 50 byte shingles. Another blank document created a day later only shares about 80% of the same shingles. This has two consequences. First, two documents that look identical to the user will still be somewhat different. Second, two documents that look totally different to a user can share close to 20 KB of header information.

On a positive note, the similarity metric performs well when Word simply opens any file and saves it with another name. Also, it performs well when large chunks of the input file are found in the output file. The metadata present in Word files is a problem when we open a plaintext file and save it as a Word document. We expected that the similarity metric would detect the text in common between the two files, and the similarity would be proportional to the size of the text compared to the size of the header in the Word document. However, this was not the case. The Word document actually inserts formatting information into the text—degrading the similarity detected considerably. It is possible to extract the text from the metadata in Word documents, but this is not a general purpose solution, there exist data formats where this will not be possible.

The results with Microsoft Paint were similar to those with Notepad. Again, the similarity metric detects when paint copies the entire file. Modifying portions of the file produces interesting results. Changing horizontal rows of pixels is equivalent to changing large blocks of text in a plaintext file. Changing vertical rows behaves the same as making many small changes spaced throughout a plaintext file. This means that the similarity detected in the first case is larger than the similarity in the second, though visually the changes look almost identical.

These results indicate that the accuracy of the similarity metric is very dependent on the type of transformation made to the data. This seems to suggest that another similarity metric could obtain more accuracy. For instance the Hamming Distance, which counts the differences in the strings, would indicate a fairly high similarity in the case where every other character was replaced. However, any statically computed similarity metric will suffer similar drawbacks to the one used here. For example, no general purpose similarity metric will be able to determine the input file that resulted in a particular compressed or encrypted file.[1]

The similarity metric performed well on the *DDD* problem given that it performs no program analysis, virtually no instrumentation, and does bookkeeping entirely offline. The particular metric we used gave True Positives when large chunks of data were present in both the input and output files. It gave False Negatives when the transformations on the data were more complicated—from slightly less similarity than expected when Word document metadata was added to no similarity at all for WinZip files. It also performs well when there was no data dispersal. In order for there to be a False Positive there would need to be a large amount of data in common due to coincidence. The headers found in files such as Word Documents can cause this situation.

The similarity metric runs entirely offline. Thus there was no runtime overhead associated with this technique at all. However, this means that offline processing overhead is incurred. The similarity metric that we have chosen takes time that increases linearly in the size of each file for an initial preprocessing step, and a small constant time to compare two files after this. Furthermore, each file only needs to be preprocessed once, after which the *sketch* of the file can be stored. This means that large numbers of files can be compared efficiently, and newly created files can be compared against the set of existing files quickly.

Others have taken advantage of these characteristics to create plagiarism detectors which store a huge database of sketches for existing work. New work can be checked for suspicious similarity with old efficiently [2][17]. This can be seen as a solution to a specific instance of the *DDD* problem. The "application" in this case are the authors creating new work. Authors certainly take existing work as input, plagiarism detectors attempt to ensure

[1]If we discovered such an algorithm, this thesis would have a different topic!

		Expected Answer	
		Yes	No
Given Answer	Yes	Large chunks of copied data True Positive	Rare, though metadata can cause this False Positive
	No	Frequent small modifications False Negative	Common True Negative

Performance

- Incurs only offline overhead
- Data analysis is efficient—linear preprocessing step and constant comparison step
- Can analyze many inputs/outputs

Figure 3.2: Risk matrix and performance summary for the similarity metric.

that this work is not present in the output without sufficient modification.

Users can determine if these characteristics are appropriate for their particular situation. For instance, the lack of a need for runtime support, the tendency to catch large areas of copied text, and the rarity of False Positives make this method an ideal fit for document plagiarism detectors. These detectors fit into the *DDD* problem if we look at the entire space between the submission of two papers as the "application". Clearly we cannot perform runtime analysis here, but similarity metrics do not require this.

3.4 Binary Rewriting

Our experimental results indicate that the binary rewriting method is not highly reliable. The assumption that most major data movement takes place using standard library call appears to be incorrect. This assumption fails even in applications that merely edit and display data and do not perform complicated computation on the data. The results reported below reflect the limits for this method, and suggest an alternative, more high level, way for using binary rewriting to address the *DDD* problem.

The binary rewriting method is limited in scope by the nature of the functions that can be hooked. In order to hook a function, we must know about it ahead of time and

		Expected Answer	
		Yes	No
Given Answer	Yes	Large Buffers True Positive	Often. Buffers are reused and don't get marked clean False Positive
	No	WinZip-like transformations, small buffers False Negative	Open and delete all True Negative

Performance

- Data analysis can be expensive compared to hooked functions
- Compared to total runtime, hooked functions rarely run
- Overall, virtually no performance overhead

Figure 3.3: Risk matrix and performance summary for the method using binary rewriting to monitor dirty buffer movement.

the tool must be able to find it. This means that it is impossible to add a bookkeeping hook to a function that is internal to an application—for one reason we can not know about all such functions, for another, the tool cannot find them.

This does not mean that the method will necessarily fail. We do know about the standard library functions and Windows API calls that initiate data dispersal. Furthermore, the tool can find these functions because they are present in dynamically linked libraries. Thus, we can add our bookkeeping hooks to them, hoping that a large amount of data movement uses these paths and thus the final result will be useful in a practical sense.

When we add these hooks we get mixed results. We detect data dispersal when we open and save a one page text file in Notepad or PopPad. Similarly, we get a True Positive when we open and save a a bitmap file in Paint. When we open the same text file, then delete all of its contents and then save it, we get a True Negative.

This is encouraging; however, there are also common cases where we get False Negatives and False Positives. For instance, if, after we delete all of the contents of the above file, we type a page of unrelated text and then save we get a False Positive. It seems that a buffer is being reused, but not properly marked clean. We never detect data dispersal in our Word or WinZip tests. We can get False Negatives in other situations as well. For instance, if we delete only 90% of the text in the aforementioned Notepad file and then save, we do not detect data dispersal.

Methods that use buffer tracking bookkeeping cannot have only False Negatives or only False Positives. A false answer indicates that data movement has been missed by the bookkeeping. Missing the movement of clean data into a previously dirty buffer will result in a False Positive, while missing the movement of dirty data into a clean buffer will result in a False Negative.

Clearly there are problems with this technique. The False results could indicate two things. Either we have not hooked all of the data movement functions, or a significant amount of data movement is taking place outside these functions. While it is certainly the case that we have not hooked every possible library function and API call that could move data in any arbitrary application, we do not believe that this omission is the cause of these results. It is not difficult to produce a list of all of the library functions that a particular application uses, and we have ensured that we have hooked all of the appropriate functions for our tests.

Debugging information also indicates that data movement is taking place outside of hookable functions. For instance, when we set a data breakpoint on the beginning of a legitimately dirty buffer and perform an action, such as replacing text, that should clean the buffer, we have observed breaks in one of three places. First, we sometimes break in functions that we have hooked. This is good, it means that we will mark the buffer clean. The other places we break indicate more inherent problems. We sometimes see breaks in unexported library functions. These are functions that are used as helpers to other library functions. User code cannot legitimately call them since they are not listed in the library export list. For instance, the API function to get the text from an edit window uses an unexported function to lookup the buffer where the text is stored and copy it into a user buffer. We clearly want to add bookkeeping to this operation, however this is not simple. First, it would involve hooking an unexported function. This is not a good idea because this function is liable to change in other releases of Windows, creating a maintenance issue. It would be nice to be able to hook the `GetWindowText` library function that calls the unexported function. This leads to problems as well. In this function, we do not have access to the buffer where the text in the window is stored. Again, we could work around this using undocumented implementation details, but this is not a practical solution. In the end, we are forced to conclude that there is some data movement that takes place in unexported library routines that we cannot practically hook.

The last place that we sometimes break is in normal user code. Some data move-

		Expected Answer	
		Yes	No
Given Answer	Yes	Whenever there is dispersal True Positive	Whenever data is read in but not sent out False Positive
	No	None False Negative	Whenever the file isn't read True Negative

Performance
• No measurable overhead

Figure 3.4: Risk matrix and performance summary for using binary rewriting solely to monitor input and output channels.

ment is performed outside of library calls altogether. For example, WinZip performs its compression routine in the application proper. This means that we cannot hook the function that performs this data movement. We are stuck with missing data movement results for these types of applications.

Although using binary rewriting to track dirty buffers seems to be unlikely to produce consistently correct answers to the *DDD*, it is likely that binary rewriting can be used as a partial solution to the problem, perhaps in conjunction with other methods. Binary rewriting provides a clean, and, as we will see, efficient method for monitoring the input and output channels of an application. As a simple example, if we report a likely data dispersal any time we see sensitive information on an input channel we will get never get a False Negative answer. Thus we can focus on using other techniques to reduce the number of False Positives.

The binary rewriting method incurs virtually no overhead. For the Notepad test this method had a median of 32 seconds, a low of 28 seconds, and a high of 35 seconds. The WinGrep test had a median time of 330 seconds, a low of 302 seconds, and a high of 337 seconds. While the median time is slightly longer for this method than the baseline, as one can see from Figures 3.5 and 3.6 this difference is not significant compared to the range of times that was seen.

This result was expected. The bookkeeping code that we add using binary rewrit-

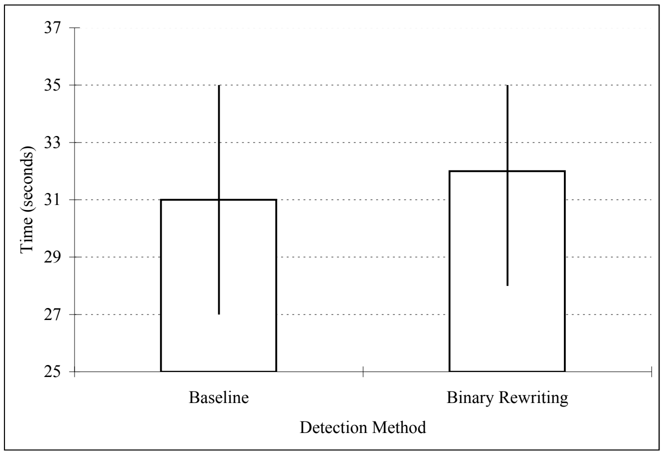

Figure 3.5: The interactive test with Notepad for binary rewriting. The bar indicates the median runtime of the test and the line indicates the range.

ing certainly could cause a significant increase to the time of some of the hooked data movement functions. For instance, `memcpy` is usually simply a tight loop that moves data from one buffer to another. Our bookkeeping code could even take longer than this movement for short buffers. Nonetheless, it is clear that the data movement functions do not account for a significant portion of the overall runtime of the applications. We add overhead to functions that are seldom called in proportion to the rest of the application code, thus our overhead is not seen when looking at the entire program runtime.

3.5 Binary Interpretation

As we do not yet have a full working implementation of the binary interpretation method, it is not possible to give a complete analysis of the risk/performance tradeoffs characteristic of the method. Nonetheless, we can use the solution parameters and our experience with the first two methods to provide an analysis of the expected results.

We saw in the analysis of the binary rewriting method that the important cases in terms of risk were those data movement functions that we had not hooked. Similarly, for this method the cases important to the analysis are those functions that we will not

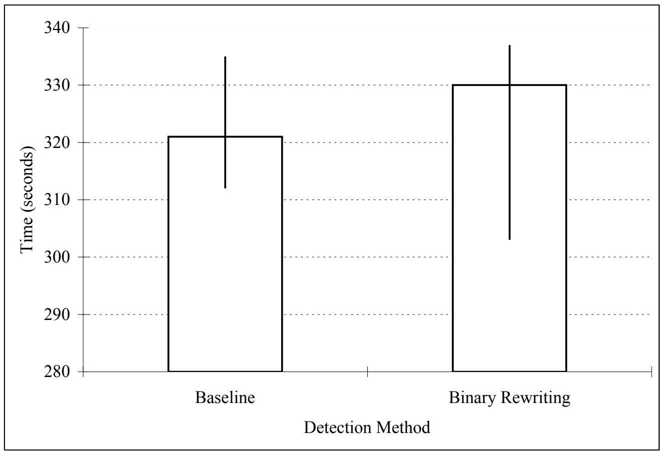

Figure 3.6: The WinGrep test for binary rewriting. The bar indicates the median runtime of the test and the line indicates the range.

instrument and those that we will instrument incorrectly.

The first clear case that this method will miss is data that is dispersed but not given to functions via arguments. This could happen with global variables or, perhaps more commonly, with an extra level of indirection. For example, a function that is passed an object and moves data from a data buffer contained in the object. Another case where this method will leak data is when a function changes behavior after it has been instrumented. Take, for example, a function that is passed two buffers and a flag to indicate which direction to copy data. This method will only perform correctly on this function when it copies data in the same direction as it did the first time it was called.

These two cases are common enough that we expect this method to miss a fair amount of data movement. However, we do expect that it will miss less data movement than the binary rewriting method. This indicates that binary interpretation is a promising avenue for risk mitigation in the *DDD* problem.

The runtime overhead associated with the binary interpretation method is much larger then that of the other methods. The interpreter running without any instrumentation code incurs a large performance penalty. For the Notepad test the median time was 49 seconds, the low was 45 and the high was 55—close to twice the times of before. The

		Expected Answer	
		Yes	No
Given Answer	Yes	Data movement by functions with buffers in arguments True Positive	Globals, non-pointer arguments, changes in function behavior... False Positive
	No	Globals, non-pointer arguments, changes in function behavior... False Negative	The rest True Negative

Performance

- Binary Interpretation can be efficient
- Program Analysis overhead will be noticeable
- Instrumentation and data analysis is similar to the binary rewriting method

Figure 3.7: Risk matrix and performance summary for the binary interpretation method presented here.

Notepad test did not finish on the 1 Gb input file, with a 500 Kb input file the test took 120 seconds.

These numbers are misleading, however. The overhead associate with the interpreter can be significantly reduced via improved engineering and optimizations. Binary interpreters exist that break-even with non-interpreted execution time [1]. Thus, a more appropriate performance measurement for this test is the difference between the overhead of the interpreter and the overhead of the interpreter while inserting instrumentation code. This overhead will still exist in a well engineered interpretation system.

We have not finished the implementation of this method, so we do not have actual performance figures. However, we expect the program analysis overhead to be high. It is expensive to handle the faults of so many write-protected memory pages. Furthermore, it seems sensible that the analysis overhead would be fairly high, as we are attempting to determine the behavior of a fairly high level unit of code.

We do not expect the instrumentation and data analysis overhead to be as high as the program analysis overhead. We are inserting the same code in this method as we do in the binary rewriting method, which has very little overhead. We do instrument more

Method	**Risk**	**Performance**
Similarity Metric	Misses data dispersal with complicated transformations	Only offline overhead, efficient data analysis
Binary Rewriting	Misses data movement not done in library calls	No measurable overhead
Binary Interpretation	Misses data movement in functions that change behavior and when buffers are passed to functions via multiple levels of indirection	Overhead for interpreter is minimal, overhead for program analysis expected to be noticeable, overhead for data analysis expected to be low

Figure 3.8: An overview of the risk and performance associated with each method.

frequently here, but only at call sites that have data movement associated with them. This expectation could be proven incorrect if we identify many more call sites than functions we hooked before. This would mean that the instrumentation that we add will be a more significant portion of the total amount of code run.

3.6 Summary

We have examined three methods in detail. Each has a characteristic way of performing program analysis, instrumentation, and data analysis that informs the risk/ performance tradeoff that it makes.

We started with a similarity metric which performs only a sophisticated data analysis. This allows the method to perform in situations where other methods cannot. Also, the similarity metric is efficient, allowing large numbers of documents to be compared. Unfortunately, these characteristics also serve to limit the types of data dispersal that the similarity metric can detect. This method will only work when the transformations that an application does to the data is minimal.

In order to address this limitation, a method must perform program analysis. The first such method that we explore is binary rewriting common data movement functions. This method used offline program analysis to identify the functions that move data. It instruments these functions with simple data analysis code that tracks dirty buffers. Again, we find that performance is not degraded with this method, but that it misses too many instances of data dispersal.

Next, we explore a method that uses binary interpretation to perform dynamic program analysis. This allows us to instrument locations, such as user defined functions, that we could not hook. This increased program analysis comes at a price. Most likely, the performance overhead of this method will be noticeable to a user. Furthermore, there are still functions, such as those that take objects as parameters that the method will not instrument correctly.

These three methods illustrate the nature of the tradeoffs that can be made while attempting to address the *DDD* problem. In the next chapter we utilize the knowledge gained by studying them to suggest potentially fruitful directions to pursue.

Chapter 4

Discussion

Thus far we have focused our exploration of methods to address the *DDD* problem in a top-down manner. We first addressed similarity metrics, then binary rewriting, and finally binary interpretation, each time lowering the level at which we perform program analysis and instrumentation to increase accuracy while diminishing performance. While these methods are suitable for certain domains, we believe that further study should be focused in a bottom-up manner, starting at a level that is perfectly accurate and raising the level of the program analysis and instrumentation in order to increase performance.

At the lowest level a method can analyze every instruction in the binary interpreter. It can then instrument any instruction that moves data to or from a memory location or register. This will mean that at any time the data analysis code will know exactly which locations in the machine contain dirty data. Unfortunately, since this method will potentially insert many instrumentation instructions per single application instruction, it will be too slow for any practical use.

We focus the following discussion on improving the performance of this method by lifting the level of instrumentation and data analysis. This will mean that we will have to perform more sophisticated program analysis, but this should be a win overall since the program analysis is done once, while the instrumentation code and data analysis is run every time the application code that it surrounds is run.

One natural starting point is to instrument only at the basic block level. Since there are no control transfer instructions in the middle of a basic block, the data movement that takes place inside is fully determined before the block is run. This means that instrumentation inserted at this level can be as accurate as instrumentation inserted at every instruction. Since the same level of data analysis would be done in this case, we will get a performance gain overall.

It is likely that this will still be too slow to use in a normal situation. The real question, then, is whether it is possible to lift the instrumentation level significantly higher, while still maintaining the required accuracy. Work has been done that uses program analysis to make data breakpoints more efficient [20]. In this work, the authors have analyzed loops in order to bound the data addresses that the loops can touch. This enables them to significantly reduce the number of instructions that they need to instrument. While the *DDD* problem is different from the problem of implementing practical data breakpoints, similar program analysis should be able to reduce the number of instructions that need to be instrumented to address the *DDD* problem in this way.

Another possible optimization would be to decrease the accuracy of the data analysis. Thus far we have focused on using the buffer tracking described in Section 2.2. This technique keeps track of buffers at a very fine granularity. It can follow any buffer, from single addresses on up. This level of data analysis can be relaxed. For example, a method could keep track of pages in memory that contain dirty data, as opposed to every buffer. This will result in less overhead for the method as a whole. However, when using this sort of data analysis, a method loses accuracy. Marking an entire page dirty when only some of the data is dirty will produce False Positives. Furthermore, this complicates the procedure needed to clean data. Cleaning an entire page when some of the data is dirty will produce False Negatives. Nonetheless, changing the level of data analysis may be an acceptable tradeoff for some users.

The previous suggestions attempt to increase the complexity of the program analysis in order to raise the level of the instrumentation and data analysis. It would also be beneficial to decrease the program analysis overhead, especially if we can do so without lowering the level of the other solution parameters. One way to do this would be by caching the analysis between application runs. Work has been done with that suggest that caching code optimizations can be beneficial for dynamic optimizers. This work indicates that significant numbers of optimizations can be reused between different runs of an application, and even different runs of different applications [8]. It seems likely that the program analysis done dynamically for the *DDD* problem could also benefit from caching between runs.

Increasing the scope of the possible solutions to this problem leads to some interesting avenues. For instance, adding certain hardware support could significantly increase the performance of a perfectly accurate method to address the *DDD* problem. One example of this is Mondrian memory protection (MMP) [21]. MMP allows multiple memory protection domains, at the granularity of a single memory address. A possible method to address the *DDD* problem would protect the just the dirty addresses in memory. Any protection fault would then be an instance where memory was being dirtied or cleaned—meaning the method would either protect or unprotect a memory location. This method does not do any explicit program analysis or instrumentation; these are made implicit by the hardware. The simple data analysis is done in the fault handler.

Similarly, work has been done that harnesses language support to address the *DDD* problem. Sophisticated type systems can be used address the problem of non-interference [16]. As described in Chapter 1, non-interference is a superset of the *DDD* problem. These type systems allow the authors of applications to ensure that data is not dispersed. In order to address the *DDD* problem as we have stated it, the authors must be able to prove to users that data will not be dispersed without being documented. Another language technique, proof carrying code, can be harnessed to address this issue. Users can mandate that any code that they run will first prove that it documents data dispersal. This system will work well as long as users do not want to run applications that do not contain proofs.

Another possible language based approach would be to interact with a garbage collector. A garbage collector maintains data about the current state of memory in a program. A method to address the *DDD* problem could potentially harness this to perform more sophisticated or more efficient data analysis. Unlike the previous methods, language based methods rely on the authors of applications to play a role in ensuring that data dispersal is documented. With this help, a new range of methods can be explored.

There are many paths that future work on the *DDD* problem could take. Working to raise the level of instrumentation in methods like those described in this thesis is promising, though it is unclear if this will result in practical methods to address the problem, or if this goal is inherently unrealistic. If so, there are definitely boundaries that can be relaxed that will open up possibilities for methods that will likely be able to address the problem in a practical manner.

Chapter 5

Related Work

There are several areas of work that relate to the techniques employed in this thesis. On a high level these areas break down into two categories. Static techniques have been employed in relation to data flow and data comparison while dynamic methods have been employed for purposes other than data analysis.

5.1 Static Techniques

Document similarity metrics have been used in many contexts. Several groups have used them to build large-scale document plagiarism detectors [2][17]. The main difficulties addressed with these systems are their scale, but they illustrate the usefulness of similarity metrics for comparing files that may have overlapping data. Even when people make some attempt to obfuscate their plagiarism, the metrics are often helpful. Plagiarism detectors are clearly a good match for the similarity metric method, as one can see by looking at our analysis of its usefulness to the *DDD* problem. It produces True Positives when large chunks of data are copied and rarely produces a False Positive– a very good match for this area. Also the offline nature of the method is necessary for this situation.

Researchers in programming languages have investigated information flow throughout programs. They have used type systems to classify data according to programmer defined policies. The compiler can then ensure that these policies have been followed. These techniques don't merely ensure data is not copied from one place to another but can also be used to ensure that certain data does not affect the control flow of the program. Thus an attacker cannot even make inferences about protected data from timing measurements.

Sabelfeld and Myers provide a good survey of the techniques in this field [16].

While these systems are extremely useful, they solve a slightly different problem than the one attacked in this thesis. Type systems can help programmers enforce a policy that they set in a large complex software system. However, they rely on those programmers to want to use them. This gives no guarantee to users that a policy that they want enforced has been. Proof carrying code promises to provide a proof that a program follows a particular policy that a user can verify independently [12]. Unfortunately there are still roadblocks to overcome before this technique can become practically useful [7].

All of these methods are data-focused, but they are static—that is they are used either before or after the program has run. They do not take take advantage of the opportunity to actually observe the data during run-time.

5.2 Dynamic Techniques

Dynamic techniques actually interact with the execution of the application. They have been used in many areas– program optimization, program sandboxing for security, and control flow profiling are among them.

Dynamo, a system developed at Hewlett-Packard Labs, uses an interpretation and code-caching system for dynamic optimizations [1]. We use the interpretation loop to watch for data flow. Dynamo uses it to identify "hot" program traces that it can optimize. Somewhat remarkably, it is able to recoup the interpretation overhead and provide a speed-up to many classes of applications.

Others have used the framework provided by Dynamo to instrument the application binary at runtime for general security purposes. "Program Shepherding" watches the control flow of a program and ensures that it only follows a set of "safe" transfers [10]. Similar attempts to confine security breaches have been made using binary rewriting techniques. Intercepting system calls and checking them against a preset policy can prevent some malicious code from doing damage. [6]

Work has also been done using binary translation for profiling purposes. Shade is a system developed to provide fast trace generation of profile data. It uses binary translations to insert tracing instructions into the instruction sequence, which is then cached for performance reasons [4].

In the same way that data flow profiling is similar to control flow profiling, data breakpoints used in debugging are similar to control breakpoints. Data breakpoints provide a narrow view of data flow. They allow a programmer to see when the value stored in a particular location in memory changes. Work has been done to make these breakpoints work at a reasonable performance cost [20].

Chapter 6

Conclusion

We have examined the risk/performance tradeoffs that any method addressing the *DDD* problem must make. We use the solution parameters, program analysis, instrumentation, and data analysis, to characterize methods and reason about the tradeoffs that they make. A top-down exploration of the method space yielded three methods that attempt to address the problem. While none of them provide perfect answers, they may provide users with the characteristics needed in a certain situation. We believe that the future search for a solution to the *DDD* problem should focus on a bottom-up approach, on methods that harness the power of some outside help such as hardware or language support.

The effects of the solution parameters on risk and performance should guide the bottom-up approach. In general, program analysis is done once, while instrumentation and data analysis is done throughout the run of the application. The implication is that performing more sophisticated program analysis in order to reduce the instrumentation and data analysis will result in overall performance gains. Using more approximate data analysis could also enhance performance but must be done carefully. Our results indicate that even small data leaks can significantly increase the risk of False Positives and Negatives.

When evaluating these methods, it is easy to lose sight of the goal of a *practical* solution to the problem. We stress that the fact that a method can produce a False Negative or Positive should not automatically disqualify it as a potentially useful method. We have seen that similarity metrics, for instance, can produce wrong answers, but have characteristics that are ideal for serving as plagiarism detectors. Binary rewriting could be used to monitor an application's input and output channels. If any output channel is ever used after the corresponding file has been read, then the method can document that data was dispersed. The result will be a large number of False Positives, but no False Negatives. The process, meanwhile, will be fast. It is possible that this would fit a users needs for a practical situation.

Documented Data Dispersal is an important problem to address. It can serve to save individual users' privacy and save corporations money. As such, it is important to develop methods that are capable of providing practical answers, even while we search for more perfect solutions.

Bibliography

[1] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices 35*, 5 (2000), 1–12.

[2] Brin, S., Davis, J., and García-Molina, H. Copy detection mechanisms for digital documents. In *Proceedings of the ACM SIGMOD Annual Conference* (1995), pp. 398–409.

[3] Broder, A. Z. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of SEQUENCES* (1997).

[4] Cmelik, B., and Keppel, D. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS Conference on Measurment and Modeling of Computer Systems* (1994).

[5] cut-the-knot.org. Distance between strings. www.cut-the-knot.org/do_you_know/Strings.shtml.

[6] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th Usenix Security Symposium* (San Jose, CA, USA, 1996).

[7] Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z. A syntactic approach to foundational proof carrying-code. Tech. rep., Yale University, New Haven, CT, January 2002.

[8] Hazelwood, K., and Smith, M. D. Characterizing inter-execution and inter-application optimization persistence. In *Submitted to ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators* (2003).

[9] Hookway, R., and Herdeg, M. Digital FX!32: Combining emulation and binary translation. In *Proceedings of Digital Technical Journal* (1997), vol. 9, pp. 3–12.

[10] Kiriansky, V., Bruening, D., and Amarasinghe, S. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (2002).

[11] Magnusson, P. S., Larsson, F., Moestedt, A., Werner, B., Dahlgren, F., Karlsson, M., Lundholm, F., Nilsson, J., Stenström, P., and Grahn, H. Simics/sun4m: A virtual workstation. In *Proceedings of the Usenix Annual Technical Conference* (1998), pp. 119–130.

[12] Necula, G. C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)* (Paris, January 1997), pp. 106–119.

[13] Pierro, A. D., Hankin, C., and Wiklicky, H. On approximate non-interference. In *IEEE Computer Security Foundations Workshop* (June 2002).

[14] PricewaterhouseCoopers, U.S. Chamber of Commerce and ASIS Foundation. Trends in proprietary information loss. Survey Report, 2002. http://www.asisonline.org/newsroom/surveys/spi2.pdf.

[15] Rojas, P. Kazaa lite: No spyware aftertaste, 2002. Wired News: www.wired.com/news/mp3/0,1285,51916,00.html.

[16] Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas on Communications 21*, 1 (January 2003).

[17] Shivakumar, N., and García-Molina, H. SCAM: A copy detection mechanism for digital documents. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries* (1995).

[18] Smith, M. D. Personal communication.

[19] Thompson, K. Reflections on trusting trust. *Communication of the ACM 27*, 8 (August 1984), 761–763.

[20] Wahbe, R., Lucco, S., and Graham, S. L. Practical data breakpoints: Design and implementation. In *SIGPLAN Conference on Programming Language Design and Implementation* (1993), pp. 1–12.

[21] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. In *Proceedings of ASPLOS-X* (Oct 2002).