IUCS-TR670
Leapfrog: Enhancing Information Protection in Commodity Applications with Dataflow Control

XiaoFeng Wang, Zhuowei Li and Rui Wang
Indiana University at Bloomington

Abstract

Commodity applications can pose a serious threat to users' confidential information when they do not have sufficient security features or are configured improperly. This problem is difficult due to the unavailability of these applications' source code, which renders the techniques such as compiler-level security enhancement hard to apply. Existing solutions rely on either system-call level control, which is often too coarse-grained, or instruction-level dataflow tracking, which is too expensive to operate online. In this paper, we present a new solution called *Leapfrog* which retrofits binary executables with mandatory dataflow control. Our technique enables a "patched" application to perform fine-grained dataflow control at a performance penalty which in many cases can be neglected. This is achieved through a novel technique that tracks sensitive data flows only at a small set of program locations: each location uses the program's internal state and pre-computed conditions to predict the path the data flows will go through and the next location they will reach. As a result, the sensitive data can be followed until they are to be sent out to the Internet, where they are controlled according to security policies. Such dataflow tracking and control is supported by an offline analysis which identifies the execution paths for processing sensitive data and the conditions for the data to propagate along these paths. We further mitigate the coverage concern of this analysis through enforcing a security policy that disallows highly sensitive data to be processed by unknown execution paths *without* disrupting a program's operations. Leapfrog works on multithreaded applications and can attach code to an application without functionally altering its executable files. Our evaluations show that our technique effectively protects sensitive information in misconfigured applications and those with security flaws, and also incurs a small runtime overhead.

1 Introduction

Nowadays, network-facing applications such as Email, browsers and peer-to-peer (P2P) software offer users unprecedented opportunities to connect to the world. Such benefits, however, should be appreciated discreetly against the potential risks these applications may bring in. Freeware and even commodity software often do not have adequate security features to protect users' confidential information, and in the cases they do, proper use of these features to manage shared resources can be a daunting task to ordinary users. As a result, confidential information is often leaked through applications. For example, a usability study of Kazaa P2P file-sharing in 2003 reveals that many users made their email inboxes and credit card numbers publically available on the Internet [21]. As another example, a Japanese police officer recently has been fired for inadvertently leaking thousands of police records to the Internet through the Winny file sharing application which shared the entire hard disk of his work computer [5]. Concerns for information leaks come from not only some free P2P software, but also the applications from renowned producers. An example is Apache 2.0 for Windows which contains a flaw allowing one to retrieve files from CGI-BIN directory [1]. A more recent report shows that IT companies are worried about the 'search across computers' feature in Google Desktop which could let their employees inadvertently disclose corporation data [4].

The problem of information leaks has been extensively studied in the research on information-flow security, which proposes to trace sensitive data in an application to prevent them from being leaked to an unauthorized party. For example, if a reliable correlation can be established between the data an FTP server reads from a confidential file and the packets it delivers to the Internet, we can prevent sensitive information from being sent to an unauthorized FTP client. This can be achieved most effectively if the application is designed with consideration of information flow policies [50]. Unfortunately, software developers rarely follow such a path, as economic concerns usually drive them to choose functionality over security. Alternatively, one can retrofit the source code of legacy software to add in security features [19]. A problem here is that many commodity applications do not have their source code publicly available and therefore these features often need to be implemented in binary executables. A technique for this purpose is instruction-level taint tracking [48], which monitors the execution of an application at a per-instruction base so as to acquire fine-grained control of its data flows [18]. A serious weakness of the technique is performance overhead, which is so high that without hardware support, it usually slows down a program by an order of magnitude [34].

To effectively control information leaks via a network-facing application, several technical challenges need to be addressed. First, a practical solution is expected to work on binary executables and make no changes to operating systems and hardware architecture. Second, it must be sufficiently fine-grained for identifying sensitive data flows. This is important because a coarse-grained approach, such as those working on the system-call level [42], could result in either inadequate or excessive control of the application's behavior, and therefore reduces its overall usability. As an example, system-call controllers such as systrace [2] and AppArmor [3] can effectively regulate which files an application is allowed to access. However, once a file has been read, they can be less effective in controlling how the data derived from the file can be used and where they can be sent. This is because without an instruction-level observation, those approaches may not be able to correlate the application's output with its input in a reliable manner. Third, the approach must be lightweight and should only incur an overhead acceptable for online operation. To this end, we propose a new technique in the paper, which we call *Leapfrog*.

Leapfrog. The objective of Leapfrog is to retrofit the binary executables of legacy applications with mandatory dataflow control so as to improve their protection of sensitive user information. This is different from the prior work [19] that works on source code and focuses on regulating the way in which a subject (e.g., processes) accesses an object (e.g., files, memory), instead of controlling the propagation of the data derived from an input. As a first step towards this goal, our current design is for preventing information leaks through a category of *data-transferring applications* which send local data to remote recipients. Examples of such applications include Web servers, FTP servers and P2P software. A prominent property of Leapfrog is that it tracks sensitive data flows only at a small set of program locations called *checking nodes* or simply *nodes*. Nodes are selected from a program's execution paths that are responsible for transferring data from local inputs to network outputs. We call these paths *data-leaking paths*, and use *hop* to describe data flowing from one node to the next one along such a path. At the program's runtime, Leapfrog first identifies sensitive data at an input node that imports them from a file system. It then checks the program's internal state to determine whether the execution will cause the next node on a data-leaking path to receive the sensitive data. As such, Leapfrog tracks a sensitive data flow hop by hop until it hits an output node where data are to be delivered to the Internet. Access control will be enforced at that node through checking the recipient's IP addresses against those of authorized parties. This tracking and control approach also works on the applications with concurrently-running threads.

Leapfrog analyzes the binary executable of an application offline to identify data-leaking paths. It selects nodes from these paths to ensure that conditions for individual hops can be easily computed using symbolic execution [24]. Such a treatment allows a dataflow track at a high level (close to call interception) to attain a control granularity close to that of an instruction-level analysis. However, this benefit comes with costs. Of particular challenge is the problem that some data-leaking paths may not be found offline, because analysis of binary executables is well recognized to be a hard problem [25]. As a result, sensitive data can still be leaked out through the execution paths which the offline analysis missed. We address this problem using two security policies which trade functionality for security: the optimistic policy permits data to flow through unknown paths to avoid undermining an application's functionality, whereas the pessimistic policy disallows data to be transferred through any unmediated path so as to protect highly sensitive information. The second policy is enforced through controlling the entry points from which data get into a program and a mandatory sanitizing operation that cleans sensitive buffers when the program will take an unmediated path. We present a novel sanitizing technique in the paper that removes the sensitive data from the memory of a process *without* breaking its execution.

Leapfrog adopts some seemingly heavyweight techniques, symbolic executions in particular. However, we only use these techniques in a very lightweight manner: our approach only verifies a set of pre-computed conditions online, which turns out to be extremely efficient. Actually, our evaluation shows that the performance overheads incurred by Leapfrog are well below 20% for most applications we tested. On the other hand, our technique is also capable of controlling sensitive data flows at a granularity approaching that of instruction-level tracking, as demonstrated in the evaluation.

Contributions. We make the following contributions in this paper:

- *Practical and fine-grained online dataflow control.* We present a new technique which achieves fine-grained tracking and control of sensitive data flows, and also works efficiently online. This technique can be applied to enhance the security features of commodity applications even when they contain concurrently-running threads. It can also be used to enforce mandatory dataflow policies for preventing information leaks caused by inadvertent misconfigurations.
- *Tradeoff between security and functionality.* We designed two security policies to mitigate the coverage problem that is deemed inevitable in analyzing binary executables. These two policies apply to the information with different sensitivity levels, at the discretion of a system administrator. We also developed a novel sanitizing technique that enforces these policies to a program without breaking its normal operations.
- *Flexibility and ease of use.* We employ a composition of static and dynamic instrumentation to efficiently attach the code for dataflow control to a commodity application without functionally altering its executable files. This gives a system administrator flexibility to modify the dataflow policies she intends to enforce, and switch on and off the control mechanism with ease.
- *Implementation and Evaluations.* We implemented our design to a prototype system using Pin, a dynamic instrumentation tool [28], and other tools we developed for symbolic execution and static binary rewriting. Our experimental evaluation demonstrates that our technique controls sensitive data flows at a fine granularity and incurs less than 20% performance degradation in most situations.

The current design of Leapfrog works most effectively on data-transferring applications though we believe that the basic idea can be more general. It remains to be an interesting question how to extend the current design to handle other applications such as Microsoft `Word`. In addition, Leapfrog aims at offering an efficient framework to track and control data flows, upon which security policies can be enforced, rather than proposing a new policy model and language. Throughout the paper, we present our technique using a set of simple policies as an example, which forbids the data derived from a sensitive document to be sent to an unauthorized IP address.

2 Design

Leapfrog takes two steps to achieve efficient dataflow control: offline analysis and online monitoring. The first step includes an *offline taint analysis* of binary executables to identify data-leaking paths and tainted

```
01    void foo[100][4096];
02    int i = 0;
03    ...
04    int senddata(int fd, int sd)
05    {
06        ...
07        l = read(fd, inbuf, 4096);
08        if(l <= 0) return -1;
09        memcpy(foo[i], inbuf, l);
10        j = input();
11        l = strlen(foo[j]);
12        memcpy(outbuf, foo[j], l);
13        j = 0;
14        if(sd > 0) status = send(sd, outbuf, l);
15        ...
16    }
```

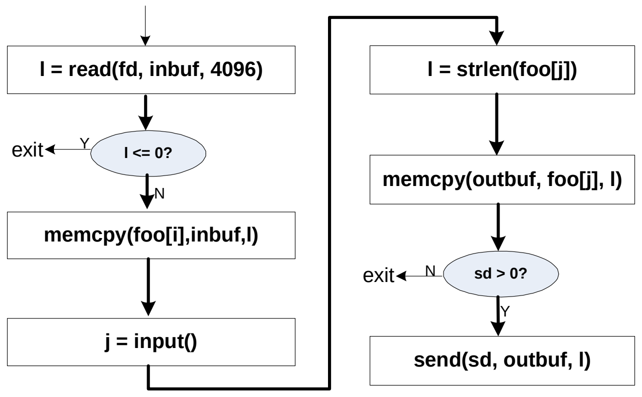

Figure 1: An Illustrative Example. Three monitor nodes are set at Line 7, 10 and 14.

buffers, *symbolic execution* to compute the conditions for sensitive data to flow through these paths and *instrumentation* to inject tracking and control code to selected nodes on these paths. The second step involves *dataflow tracking* that monitors sensitive data at individual nodes and verifies the conditions to predict the next hop they will go through, and *policy enforcement* that implements security policies at these nodes. These steps and their components are elaborated in this section. We also implemented a prototype system using Pin [28] to evaluate this design, which is described in Appendix.

2.1 Overview

The idea behind Leapfrog can be explicated using a toy example in Figure 1. The example is in `C` for illustration purpose. Leapfrog actually works on binary executables. The program in the example first reads data into `inbuf` and then moves them to another buffer pointed by the ith entry of a buffer list `foo`. After that, it waits for an input j to copy the content of the jth buffer on the list to `outbuf`, from which the program sends data to the Internet.

Our offline analyzer first uses instruction-level taint tracking to discover the data-leaking path of the example[1] as illustrated in Figure 1. Then, Leapfrog runs symbolic execution over the path to find the conditions for tainted input data to propagate. These conditions are described by the expression: $(l > 0) \wedge (i = j) \wedge (sd > 0)$. To control data flows online, we set an input node (Node 1) at Line 7 to identify sensitive inputs, and an output node (Node 3) at Line 14 to regulate the data to be sent to the Internet. A problem here is that the program's states at these nodes do not contain sufficient information for tracking sensitive data, as we need the value of j to determine whether the sensitive data will reach Node 3. Therefore, an additional node (Node 2) is set at Line 10. These nodes are instrumented with code for dataflow tracking and control[2].

During the runtime of the program, our online monitor at Node 1 detects sensitive data from input and verifies the condition $(l > 0)$ that ensures the data will reach Node 2. At Node 2, it checks the condition $(i = j) \wedge (sd > 0)$ to ensure that the data will flow to Node 3. Dataflow control happens at Node 3 to prevent the data from being delivered to an unauthorized party. We can also clean the buffers holding highly sensitive data at Node 3 if we are not sure how the program will handle these data after Line 14.

Assumptions. We made the following assumptions in our research.

Legitimate application. The objective of Leapfrog is to enhance information protection in a *legitimate* application. Our current design cannot withstand the attack from the application under protection: it

[1]This can also be done through a static analysis.

[2]In this example, the instrumentation is simply an API wrapping, as all these nodes contain API calls. In general, we inject dummy calls to the nodes not holding APIs.

is possible for a malicious program to bypass our instrumentation and compromise security policies. We also assume that the application does not contain self-modifying code, which may cause inconsistency between its executable files and runtime image. The technique we present in the paper is tuned to data-transferring applications. However, the idea behind it can be extended to other applications, which we discuss in Section 4.

Control-flow hijacking. We assume that the application's control flow has not been compromised. Leapfrog is not meant to be used for detecting control-flow hijacking attacks, as some taint-tracking approaches do [34]. We can adopt other techniques such as address-space randomization [43] or control-flow integrity [9] to counter this threat.

Benign operating system. We assume that the operating system (OS) the application runs in is benign. Otherwise, a malicious program with a sufficient privilege can always tamper with the security policies or even the executables of the application. In addition, we also rely on OS to protect the sensitivity labels associated with individual files (Section 2.6).

2.2 Offline taint analysis

Offline taint analysis is aimed at identifying data-leaking paths and tainted buffers. A data-leaking path is an execution path in a program which imports data from a *data source* (or simply *source*), processes them and exports them to the Internet. Examples of data sources include files and memory caches. To extract data-leaking paths from a binary executable, Leapfrog first runs the program on a set of training inputs. These inputs can be gathered from service requests received by a server program during its normal operation, for example, a trace of real HTTP flows, or generated from the specifications of the application protocol the program uses to communicate with clients. During its execution, the program is monitored by a dynamic analyzer that taints the input data from a file or a memory source and tracks them instruction by instruction in accordance with a set of taint-propagation rules. The rules used in our research are similar to those adopted in other taint-tracking techniques such as RIFLE [44], TaintCheck [34] and LIFT [35]. Examples are presented in Appendix.

The analyzer exports an instruction sequence once it hits an API or system call that sends tainted data out of local system. This instruction sequence is deemed as a data-leaking path. Besides reading from file system, a data-leaking path can also transport the data tainted from a prior transaction to network. In this case, the memory area holding these data is marked as a source. For example, Apache implements a memory cache to keep the content from a prior service; the cache can be used as a source when its content is requested by a client.

Data-leaking paths are not the only output of the taint analyzer. It also labels the instructions on the paths which contribute to the observed taint-propagation process and identifies the buffers tainted by these instructions. These instructions and buffers are further analyzed using symbolic execution to prepare for a mandatory sanitizing operation that enforces a security policy (Section 2.6). The analyzer also generates information to help the follow-up analysis. An example is the relations between memory aliases.

Coverage. Dynamic analysis has a coverage problem: it cannot guarantee to find all data-leaking paths, which could allow sensitive data to be leaked out through unknown paths. A solution to the problem can be static analysis that offers techniques for "chopping" a program [37] to identify all the paths between a source and an output node. However, static analysis has only limited capability in handling binary executables, and can be easily defeated by code obfuscation which has been increasingly implemented for copyright protection [45]. Alternatively, we can apply recently proposed approaches to dynamically explore multiple execution paths [29, 11]. Though all these techniques can help improve the coverage of the offline analysis, none of them provides the guarantee to capture every data-leaking path in a generic case.

In our research, we mitigate the coverage problem through enforcing different security policies to the

```
         mov dword ptr [ecx], eax
         mov dword ptr [edx], ebx
         cmp dword ptr [ecx], 0x01
         jne loc_9
         add eax, 0x04
         mov dword ptr [ecx], eax
         mov ebx, dword ptr [edx]
         jmp  loc_10
  loc_9: sub eax, 0x04
 loc_10: ...
```

Figure 2: Code snippets for the illustration of path conditions and taint propagation conditions.

information with different sensitivity levels. For the data of which confidentiality is crucial to the user, a pessimistic policy forces them to be processed only through known data-leaking paths. This is achieved through a mandatory sanitizing operation that cleans a program's internal state of sensitive data without jeopardizing its normal execution. For other data, we could permit them to be handled by unknown paths in a legitimate program, as such a program will not automatically leak out sensitive data, and we do not want to undermine its functionality. Enforcement of these policies are elaborated in Section 2.6.

2.3 Symbolic execution

Symbolic execution works on the outputs of the dynamic analyzer to compute the conditions for a program to transfer data through a data-leaking path. These conditions include a path condition that ensures a correct execution of a data-leaking path, and a *taint-propagation condition* (TPC) that ensures data to be propagated by the execution. Moreover, symbolic execution also generates symbolic descriptions for the buffers accommodating tainted data, which are used for online control.

Path condition. A path condition (PC) is the accumulation of the conditions for determining a unique execution path through a program. These conditions contain the constraints for conditional branches on that path. The path condition for a data-leaking path can be identified using symbolic execution [24]. Symbolic execution is a technique that evaluates the execution of a program through interpreting its instructions, using symbols instead of real values as inputs. To analyze a data-leaking path, Leapfrog first uses symbolic variables to describe the state of a program at an input node, which includes register values, memory contents and input data, and then executes instructions on the path on these symbols. Such execution involves static evaluation of memory updates, arithmetic operations and branch predicates of the program, and yields a symbolic formula at every step. As a result, we obtain a set of symbolic Boolean expressions at an output node, which describe the constraints for following the data-leaking path. The path condition is a conjunction of these constraints. As an example, the condition for the program in Figure 1 is $(l > 0) \wedge (i = j) \wedge (sd > 0)$.

A raw execution path extracted from a binary executable usually contains the instruction sequences for nested API calls, which could make a path condition very complicated. This problem has been solved in our research through bypassing these instruction sequences, and instead inserting nodes to intercept the return values of the calls that are related to the path condition for the remaining path[3]. We also set nodes to other locations necessary for tracking and controlling data flows, on which we elaborate in Section 2.4. An example is the instructions responsible for reading from and writing to a shared buffer between threads. Rather than generates a unified condition for the whole execution path, our approach actually produces a set of conditions, each of which corresponds to a path segment between two neighboring nodes.

[3]An assumption we made here is that an API function always propagate the taint as observed during the taint analysis. This assumption can be released by using the model of that function.

A challenge for computing path condition is memory aliasing: a symbolic expression may point to some unknown memory address, and two expressions may point to the same address. The aliasing problem makes it difficult to determine the constraints for conditional branching, which is illustrated by the example in Figure 2. In the example, the conditional jump at Line 4 depends on the outcome of the operation that compares the data stored in address `[ecx]` with $0x01$. From Line 1, we know this address has been written to with the data in `eax`. However, we do not know whether the instruction at Line 2 also changes the content of that memory, as `ecx` and `edx` may have the same values, referring to the same memory location. Let s_{ecx}, s_{eax}, s_{edx} and s_{ebx} be the symbols assigned to `ecx`, `eax`, `edx` and `ebx` respectively. If $s_{ecx} = s_{edx}$, then the path condition is $s_{ebx} = 0x01$; otherwise, it becomes $s_{eax} = 0x01$. Our solution to the aliasing problem is based on the runtime information provided by the taint analysis, which can reveal the relations among different symbols such as $s_{ecx} = s_{edx}$. This approach handles legitimate programs well: we discovered in our research that the relations among memory aliases in a legitimate program rarely change with inputs.

Leapfrog performs symbolic execution over an instruction sequence. This treatment, however, could make a path condition specific to our training inputs in the case that an execution path contains a loop of which the number of iterations depends on inputs. To mitigate the problem, we recover loops from the instruction sequence using the repeated instruction patterns observed from the same addresses to analyze their effects on path conditions. Loop is a well-known obstacle to symbolic execution. A standard solution is the threshold approach [12] that symbolically runs a loop body for at most n times, where n is a threshold, to compute symbolic values for variables. This approach only approximates the value ranges of variables modified by the loop body, as the problem of finding their exact values in the general situation is undecideable in static analysis. This problem, however, can be mitigated with the aid of runtime information: for a loop that modifies the variables that affect a path constraint outside the loop, we can instrument its exit to acquire the values of these variables when the program is running.

Taint-propagation condition. Following the instruction sequence of a data-leaking path does not guarantee to propagate tainted data as observed in the training rounds. This problem is also caused by memory aliases, as we illustrate in the example in Figure 2: suppose that the content of `eax` is tainted; the taint will be propagated to `ebx` only when `ecx` and `edx` point to the same memory location. Such a condition is described as taint-propagation condition (TPC) in our research.

A TPC contains the relations among memory aliases which allow taints to propagate from an input node to an output node. Like our solution to the aliasing problem for path condition, such relations can also be revealed by the dynamic analysis. Specifically, the taint analyzer records the history of memory accesses including all the memory addresses to a log. From the log, we can discover a relation between a pair of symbols as follows: for an instruction that reads from a memory symbol, we search the log for the most recent write to the same memory; if the write happens to a different memory symbol, an equality relation is established between these two symbols as they all address to the same memory. The TPC is a conjunction of such equality relations[4]. For example, the TPC for two instructions in lines 6 and 7 of Figure 2 is $s_{ecx} = s_{edx}$, where s_{ecx} is the symbolic value in `ecx` and s_{edx} stands for the value in `edx`.

Tainted buffers. We also run symbolic execution to compute the symbolic expressions for the address and size of a tainted memory buffer. These expressions are used during online monitoring to determine the memory area that stores sensitive data. A major problem here is loops: for example, a loop iteratively moving a DWORD to a buffer can make it hard for symbolic execution to determine the total amount of data having been processed. Our solution is instrumentation of the exit of such a loop to directly get the number of iterations when the program is running.

[4]In our research, the smallest unit a symbol can represent is a byte. For simplicity, our current implementation does not consider the overlapping between two symbols, which may happen as a result of bitwise operations.

2.4 Instrumentation

To retrofit an application with new security features, we need to instrument its executables. Instrumentation is designed in our research to achieve three objectives: (1) effectively tracking and controlling sensitive data flows, (2) minimizing overheads for online operations, and (3) retaining the flexibility to easily turn off these new features. To this end, we must choose right program locations to instrument and a right instrumentation strategy, which are described below.

Node selection. We first identify a set of nodes that are important to dataflow tracking and control. Of particular interest to us are following program locations:

- *I/O operations.* To identify sensitive input data, we need to intercept the API calls that read from local file system and the instructions that input from a memory source. To prevent these data from flowing into unauthorized remote parties, we must control the calls for network operations. Therefore, the instructions responsible for these activities should be set as nodes and instrumented.
- *Locations for evaluations of PCs and TPCs.* Dataflow tracking in Leapfrog relies on evaluating predetermined path conditions and TPCs. This can only be done at specific locations on an execution path where information for computing these conditions is available. For example, in Figure 1, the path constraint $i = j$ cannot be evaluated until the program receives the input for j at Line 10. These instruction locations should be set as nodes.
- *Loop exits.* Instrumentation of a loop exit reduces the burden of approximating the effect of a loop body on the variables related to path conditions or tainted buffers, as discussed in the prior section.
- *API call sites.* As discussed before, we treat API functions as black boxes and instrument the locations of the calls of which the return values affect path conditions or TPCs.
- *Locations for inter-thread communication.* To trace sensitive information across the boundaries of threads, we need to monitor the communication between those threads, which usually does not go through system calls or even API calls. Our solution uses the offline taint analysis to identify the instructions in one thread that move tainted data into a shared buffer, and the instructions in another thread that read these data. The locations of these instructions are instrumented to catch data flows between the threads during the program's runtime.

Instrumentation strategy. To inject code to a node, we can choose either static binary rewriting [39] or dynamic instrumentation [28]. Static binary rewriting modifies an application's executable files to add new instructions, which introduces a very small runtime overhead. However, this technique functionally alters the application, making the attempts to remove instrumented code painful and error-prone. In contrast, dynamic instrumentation attaches new instructions to the application during its runtime and therefore does not need to change its physical files. A problem is that such a technique usually works efficiently only at intercepting invocations of library functions. To reliably instrument arbitrary program locations, most dynamic instrumentation tools [28, 33] need to do instruction translation, which is very time-consuming. Though this process can be optimized using instruction caching, its performance impacts are still significant [28, 36]. For example, the Just-In-Time (JIT) mode of Pin can double the execution time of some applications even when these applications are running without any instrumentation [28].

Leapfrog takes an instrumentation strategy to strike a balance between performance and flexibility. Our approach first attempts to instrument existing API calls to track and control data flows. In the case that a node contains no API call and no nearby call site can be used, we statically instrument that node with a dummy API call. The dummy call is chosen carefully to avoid modifying an application's functionality. An example is `close(-1)`, which we used as a default dummy call in our implementation. All the API calls, including the dummy calls, serve as placeholders for API wrappers developed using dynamic instrumentation. These wrappers carry code for dataflow tracking and control and can work efficiently online. This instrumentation strategy has a negligible impact on an application's performance and also offers the convenience to detach our code from the application without causing observable side effect.

2.5 Dataflow tracking

A program patched by Leapfrog can identify sensitive input data, and track them and the data derived from them at individual nodes. Our design also enables a reliable monitoring of the sensitive data flows among threads. The details of these techniques are described below.

Tracking sensitive data flows. Since Leapfrog tracks sensitive data based on pre-determined data-leaking paths and conditions, it does not have to keep a large set of shadow values to record the origins of the data in individual registers and memory locations, as most instruction-level tracking techniques do [16, 34]. Instead, our approach assigns each data-leaking path a unique path number and maintains a tracking table at each node to map the path passing through the node to the path condition and TPC for reaching the next node along that path. During the program's runtime, our code at an input node identifies sensitive data, verifies the conditions in its tracking table to predict the potential paths the data flow may follow, and records the numbers of these paths to an *active set*. The code at the next node continues to evaluate these conditions and removes from the active set the paths of which the path conditions or TPCs are no longer held. At an output node, security policies are enforced if the active set is nonempty.

Multithread communication and memory source. The presence of multithread communication and memory source complicates the effort to track sensitive data flows. As an example, consider a thread reads sensitive data and passes them to another thread through shared memory, and the second thread continues to transfer them to the Internet. To control the data flow in the example, it is insufficient to predict the data-leaking path the data will follow because there could be many threads sending data and we need to know exactly which one of them carries the sensitive data. Moreover, in the case that a program reads from a memory source such as a cache, we need to know whether the data being accessed are sensitive for determining when to activate the tracking mechanism.

In order to reliably track the data flows between threads, we instrumented the code which threads use to communicate with each other. Specifically, a node can be set to the program location where instructions move data to a memory area shared between threads so as to identify the buffer holding sensitive data. This can also be achieved through interposing on an API call close to these instructions such as `pthread_mutex_lock()` that locks the mutex of the shared memory to locate the tainted buffer according to its symbolic descriptions (Section 2.3). The address and size of such a buffer are saved to a record in shadow memory [32]. We also injected a node to mediate a read from the shared memory: when a thread inputs from the memory, our code at that node checks the address of the target against the records in the shadow memory, and initiates a tracking process once it is found to be sensitive. The record for a sensitive buffer will be removed when the buffer is released by the program. The same approach has also been applied to guard a memory source identified in the offline taint analysis to keep track of the data emitted from it in the future transactions.

2.6 Policy enforcement

Leapfrog enforces security policies at individual nodes to control sensitive data flows. This has been achieved in our research through a suite of novel techniques we developed, which are presented in the section. We also discuss how to assist users in classifying their documents into different sensitivity levels.

Security policies. Security policies specify where sensitive data are allowed to be sent and how such a transfer can happen. Our current policy design is very simple. We first label each file with one of the three sensitivity levels: very high ("highly sensitive"), high ("sensitive") or low ("public"). Data from a sensitive or highly sensitive file are only allowed to be transferred to the IP addresses or domains documented on a whitelist which is maintained by an authorized party such as a system administrator. Leapfrog also provides a pessimistic policy to further restrict the manner in which an application can use the data originated from a highly sensitive document: such data are only allowed to be processed by the data-leaking paths with proper security mediation. This policy is designed to protect the information extremely important to its owner from being leaked out through the data-leaking paths the offline analysis missed.

The policy for regulating destinations of sensitive data flows is enforced at output nodes, where our control mechanism intercepts the API calls for data transfer to check whether recipients of the data have proper authorization to do so. Enforcement of the pessimistic policy relies on two techniques: *entry control* and *mandatory sanitizing*, which we describe below.

Entry control. Entry control prevents highly sensitive data from getting into an application via the program locations not known to be on any data-leaking path. This treatment eliminates unmediated interactions between the application and file system. To implement the mechanism, we wrapped all the API functions the application can use to read a local file, which allows us to force the data from a highly sensitive file to go through the input nodes on known data-leaking paths.

Mandatory santizing. To prevent a data flow from deviating from mediated paths, we design mandatory sanitizing, an operation for cleaning tainted buffers. This operation can be performed by every node on a data-leaking path to control the highly sensitive data already inside an application. The idea is to erase those data whenever we discover that further execution of the program will certainly move away from identified data-leaking paths. To prepare for the operation, Leapfrog maintains a list of tainted buffers, on which each node posts the addresses and lengths of the memory areas to receive critical data before execution reaches the next node. Such information is computed through replacing individual symbols in the symbolic descriptions of tainted buffers with the concrete values obtained from a program's internal state at a node. Once our code at a node concludes that the path condition or TPC for the next hop will not be satisfied, it performs the operation by cleaning the buffers on that list except those marked as internal sources.

A critical question here is how to carefully design the sanitizing operation to ensure that it does not break an application's execution. This is achieved in our research through computing an internal state that the execution will reach should the application read from a "clean" file instead of the one with sensitive information. Specifically, during the offline analysis, we designate a symbol to every byte in an input buffer and then symbolically evaluate the execution of the application. As a result, we can precompute a symbolic expression for every tainted memory location and register, and a constraint for every tainted branch condition. To perform a sanitizing, our instrumentation at a node solves the constraints to generate new values for the input bytes that affect path conditions, zero out other input bytes that are unrelated to the execution path, and overwrites other tainted locations with the values computed by replacing the symbols in their symbolic expressions with the new input data.

As an example, consider the program in Figure 3. Let $(B_0, B_1, \ldots, B_9)$ be the ten bytes in the input buffer `buf1`. At Line 2, we discover that $B_0 >$'e' is a path condition the input must satisfy. The operations at Line 3 and Line 4 move the ten bytes in `buf1` to `buf2`, and increase their values by one: `buf2[i]`$= B_i + 1$ where $0 \leq i \leq 9$. Therefore, a sanitizing that happens at Line 5 can set `buf1` as ('f', 0, ..., 0) to meet the condition at Line 2, and as a result, `buf2` becomes ('g', 1, ..., 1) according to its relation with `buf1`. This treatment ensures that the execution of the program will not corrupt because the state is valid in a sense that it is the correct state if the input was ('f', 0, ..., 0). On the other hand, the sensitive content of `buf1` has been removed. A limitation of this approach is that the path condition still leaks out some information regarding the input. This is not an issue in practice because the data an application intends to transfer rarely taint a path condition. The performance impact of this sanitizing operation is also moderate, because constraint solving can be conducted offline, and the operation will not be conducted until Leapfrog encounters an unknown path and the data being processed happen to

```
l = read(fd, buf1, 10);
if(buf1[0] > 'e') {
  memcpy(buf2, buf1, l);
  for (i=0;i<=9; i++) buf2[i]++;
    …
}
```

Figure 3: An illustrative program for data sanitization.

be highly critical.

Multithreaded applications pose yet another challenge to the enforcement of the pessimistic policy: as a thread drops important data to a shared buffer, our code observing this action does not know whether an unmediated thread will pick them up. This problem is caused by the nondeterministic interactions among threads. It is solved in our research as follows. Our control mechanism embedded in the sender thread does not move highly-sensitive data to a shared buffer, and instead, it puts in the buffer dummy data that are made to be consistent with the thread's path conditions. The real data is kept in Leapfrog's shadow memory and can be acquired by a knowledgeable thread with a proper instrumentation. This treatment inhibits an unmediated thread from accessing the highly sensitive information[5]. It could also maintain the correct execution of the thread that works on the dummy data, as the data is consistent with the sender thread's execution. One exception is when the sender is meant to have multiple threads receive the same data. The implication of our technique here can be application specific and needs further investigation.

Classification of documents. Leapfrog requires classifying documents into different sensitivity levels. This process can be facilitated with the aid of automatic tools. In our research, we adopted an approach similar to the technique described in [27]: we developed a program that automatically infers the sensitivity levels of documents according to their discretionary access control information. For example, under Linux, we can label all the files that only their owners are allowed to read as highly sensitive, those that only owners and group members can read as sensitive and the rest as public. Other information can also be used for classification. For example, a file with a "`.pst`" extension under Windows is deemed highly sensitive. After automatic classification, Leapfrog also allows an authorized party to adjust the sensitivity levels of documents.

3 Evaluation

In this section, we describe our experimental study of Leapfrog. The study is devised to understand two key perspectives of our technique: effectiveness in protecting sensitive information and performance. To this end, we evaluated our prototype using 6 real and 1 synthesized applications, and performance benchmarks such as ApacheBench (ab) [8]. The outcomes of the experiments are reported below.

Our experiments were carried out on two Linux workstations, one as the server and the other as the client. Both of them were installed with Redhat Enterprise 4. The server has a 2.40GHz Core 2 Duo processor and 3GB memory. The client has a Pentium 4 2.53GHz processor and 1GB memory.

3.1 Effectiveness

The effectiveness of Leapfrog was evaluated with 7 applications presented in Table 1. They were retrofitted by our prototype to add in mandatory dataflow control. Using these patched applications, we studied several important properties of our technique, which include tracking and controlling the data flows from

[5]It is important to notice here that the unknown thread will not retrieve the data from the shadow memory because we are dealing with legitimate applications.

Table 1: Effectiveness evaluation of our LeapFrog prototype. Apache is tested with version 2.2.16

Name	**Type**	**number of paths**	**number of nodes per path**	**trace & ctrl**	**breaking program after sanitization?**
Apache+cache	web server	5	59,59,2,1(cache),1(sendfile)	Success	No
Apache	web server	3	6,2,1(sendfile)	Success	No
pserv	web server	1	2	Success	No
Pure-ftpd	ftp server	1	2	Success	No
Samba	file sharing server	1	2	Success	No
Napster	peer to peer	1	2	Success	No
Synthesized	multi-thread	1	8	Success	No

file system and memory cache, monitoring multithread communication, and enforcing security policies through the operation such as mandatory sanitizing.

Offline analysis. To extract data-leaking paths, we ran the client programs of those applications to download a set of documents the applications mediated. These documents had various sizes, ranging from 255 bytes to 128k bytes, which enabled us to detect different execution paths for transferring those files. The experiment was designed in a way that the same document had been downloaded for 10 times. This helped our taint analyzer to identify memory cache. After acquiring data-leaking paths, our prototype continued to generate PCs and TPCs, select nodes on the paths and instrument these nodes to embed security mechanisms.

For most of those applications, instrumentation of some of their API call sites was sufficient for tracking and controlling data flows. An exception is the Apache server with caching module, of which the data-leaking paths involve several loops based on two instructions "`rep`" and "`repne`". In our experiments, we found that four of these loops affected path conditions, and instrumented the exits of them by inserting a dummy call `geteuid()`[6]. Our wrappers identified the values of important registers affected by the loops, in particular that of `ecx` which counted the number of iterations.

Control of file downloading. During the experiment, we adjusted the sensitivity levels of files, and updated whitelists maintained by the output nodes of individual applications to add or remove the client's IP address. Our experiment involves running a client program to download files from the server. Such an attempt was successful for all the applications when the files were public or the client was authorized to do so. On the other hand, transfer of sensitive data when the client was not on the whitelist was always blocked. Following we elaborate the experiment using the base Apache and Pico server as examples.

Our offline analysis of the base Apache (without the caching module) discovered three major data-leaking paths. For the files larger than 255 bytes, Apache directly sends it to the Internet using the API `sendfile` which binds a file descriptor with a socket. This path is extremely simple, having only a call instruction. It can be turned off by setting "`EnableSendfile=off`" in `httpd.conf`. When this happens, Apache transfers large files through another path, which starts with `read` and ends with `write`. Another path is the one for sending the files smaller than 256 bytes, which uses `mmap64` to read a file and `__libc_writev` to deliver its content to network. Our prototype instrumented the API calls on these paths and generated PCs and TPCs for individual hops (Section 2.3). Control of the data flows via the first path was trivial: our code at the only node, `sendfile`, could swiftly decide whether the data can be sent to a web client. The control became complicated for the path handling small files: there were totally six hops on that path, and the first hop, the most complicated one, contained 189 conditions. In the experiment, we found that our code at individual nodes on all these paths accurately identified and tracked sensitive data flows, and successfully controlled them according to security policies at output nodes.

Pico HTTP server v3.3 has a *directory traversal* vulnerability [6], which allows the attacker to read

[6]Apache wraps most API calls. This prevented us from using the default call `close(-1)` because we could not find that function from its Procedure Linkage Table (PLT).

the files outside the html directory. This will happen if the client requests a file such as "`./../././.. /passwd`" [6]. Though Leapfrog did not fix this bug, it protected the sensitive files outside the html directory from being downloaded by an unauthorized client. This was achieved through instrumentation of the server's data-leaking path which reads from file system using `_IO_fread_internal` and sends to network via `__send`. To evaluate the patched server, we used `konqueror` browser to request a file outside the html directory `/usr/local/var/www`. We chose this browser because we found that other modern browsers such as Internet Explorer and Firefox could block the attack request. In the experiment, we observed that the attempt to download the sensitive files outside the html directory was foiled by our dataflow controller.

Memory cache. The Apache server installed with `mod_cache` and `mod_mem_cache` modules can cache web content in the memory. We set the size of the cache as 4 KB and analyzed the server over training inputs. Our taint analyzer identified five data-leaking paths. Two paths handle the files larger than 4 KB: one through `sendfile` and the other using `read`. These paths do not cache web content and worked in the same way as those in the base Apache. The paths for medium file (between 256 bytes and 4 KB) and smaller files (below 256 bytes) both use `mmap64` to import data and `writev` to output them to the Internet. They saved the content of a file to memory cache through `memcpy`. The last path is for transferring cached content. It contains a single call `__libc_writev` that wrote cached data to network. Our prototype instrumented all these paths with code for dataflow tracking and control. Under the probe mode of Pin, we observed during the server's runtime that sensitive data were tracked to the buffer caching them, the location and size of the buffer were recorded at the node `memcpy`, and the follow-up transactions reading from that buffer were detected and controlled by the code at the node `__libc_writev`.

Correctness of mandatory sanitizing. We thoroughly evaluated our sanitizing technique in our research to study its impact on a program's correct execution. For every application in Table 1, we performed a sanitizing operation at every node on every data-leaking path we discovered. In our experiment, an application first read a file marked as *highly sensitive*. When the execution reached the node set to do sanitizing, it first checked the sensitivity level of the data and then cleaned the program state as described in Section 2.6. To understand whether such an operation would compromise the execution, we continued to run the application for a while to download *public* or *sensitive* files. All these transactions turned out to be successful. This study was conducted on every node we instrumented, including those managing multithread communications. We did not observe any runtime errors and anomalous operations from the applications.

Multithreading. Most Linux applications have simple thread architectures: for example, the Apache server uses a single thread to serve a whole transaction. However, the problem of multithreading is much more serious in Windows, as thread is the building block for most of its applications. To understand the efficacy of our technique in the presence of multithread interactions, we developed a Linux program which involved two "producer" threads reading data from files to shared memory and two "consumer" threads sending the content of the memory to network. Our offline analysis captured this interaction and instrumented the API `read` which a producer employed to move data into a shared buffer, and `send` which a consumer used to move data out of the buffer. In our experiment, we let the program simultaneously read from a public file and a sensitive file. Our code on the producer's front marked the buffer receiving sensitive data by recording its location and size. This enabled the code on the consumer's front to determine whether the thread was working on the sensitive data. We observed that the sensitive data was successfully tracked across the thread boundaries in this way and controlled in the thread attempting to send them.

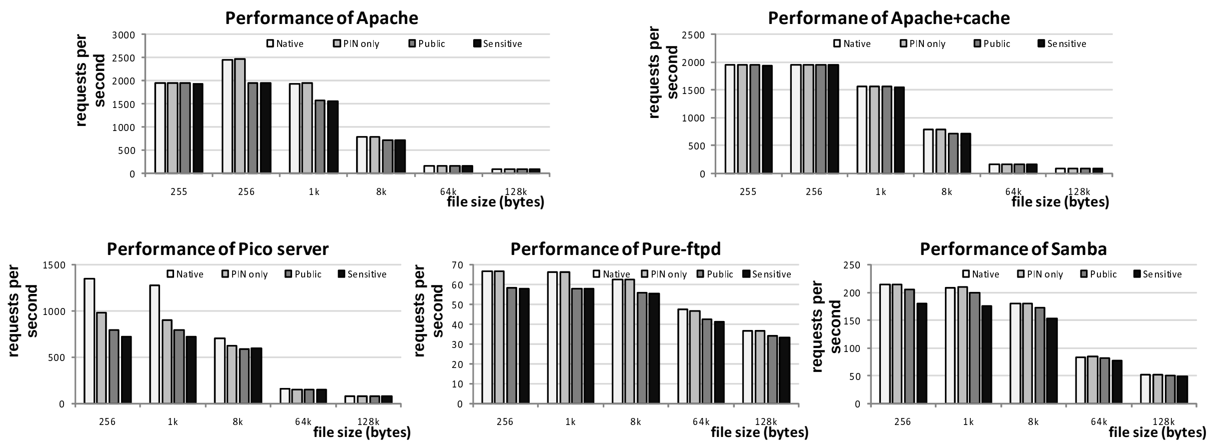

Figure 4: Runtime performance of LeapFrog. The four bars in a group represent the performance (number of requests being processed every second) of an application under the following four settings: it is unpatched ("Native"), running under PIN without instrumentation ("PIN only"), patched and working on public files ("Public"), and patched and working on sensitive files ("Sensitive"). In the experiments, we evaluated two data-leaking paths of the base Apache (without caching): one for transferring the files smaller than 255 bytes and the other for larger files, and three paths of the Apache with caching module: small file (below 256 bytes), medium files (255 bytes to 4 KB) and large files. We turned off the "`EnableSendfile`" switch, as the "`sendfile`" path contains only a call site and its performance has no difference from that of the "PIN only" setting.

3.2 Performance

We also carried out experiments to evaluate the performance of Leapfrog during runtime dataflow control and offline analysis. The offline overheads were gauged with the average time for accomplishing key analysis tasks such as taint analysis and symbolic execution. The study shows that our technique not only achieves a high runtime performance, which incurred an overhead well below 20% for most applications, but also works quite efficiently offline. We elaborate the experimental results below.

Runtime performance. We measured the performance of the Leap- frog patched HTTPds (Apache with caching, Apache without caching, Pico server) using `ab`, the standard Apache benchmarking tool [8]. For each application, we ran `ab` on the client to generate 20000 requests to download a test file from the server. Multiple tests were conducted over the file with the size ranging from 255 bytes to 128k bytes. In each test, we recorded the number of service requests these applications could handle both when the file was sensitive and when it was public. The outcomes of the tests were compared with the baseline, the performance of the original versions of the applications measured under the same test settings. Figure 4 illustrates the experimental results.

From the figure, we can see that transfer of a file through the patched applications incurred small overheads (less than 20%) in most situations. In particular, the overheads went well below 10% when the file sizes approached 64 KB. This size actually more realistically describes a real document. For example, a typical size of a web page with associated files such as images is around 100 KB. The capabilities of the patched Apache servers to serve request decreased by merely 3% in most cases except when it was transferring very small files (256 bytes, 1 KB and 8 KB). This performance was achieved even when dataflow tracking involved verifying a large number of PCs and TPCs.

An exception to the above observation is the test of Pico server over small files (128 bytes - 1k bytes), which incurred about 50% performance degradation in the worst case. This could be caused by the small size of the server which makes the impact of API wrapping observable. In addition, the probe mode of Pin we used to wrap API calls also contributed to this delay, which incurred about 30% degradation even without any API wrapping, as described in Figure 4. The problem was further aggravated when the small files was used in performance test because the server in this case had to execute the instrumented code at a very high rate. This explanation was confirmed when we were testing the server over a large file: the degradation we observed this time went well below 10%.

A similar performance was reported by LIFT on Apache server [35], which achieves 6.2% throughput degradation. However, LIFT requires hardware support to get a high performance: it runs a 32-bit program on a 64-bit system so as to use the system's extra integer register and needs 12.5% of 32-bit virtual memory as the program's shadow memory. In contrast, Leapfrog works on 32-bit systems and used around 40MB memory, less than 10% of that of LIFT. Moreover, LIFT's high performance depends on a set of optimization strategies, which work exceptionally well on Apache, but less so on other applications. On average, it introduces a slow-down factor about 3.5. Finally, while LIFT did well on throughput, which depends on network setting, choices of a performance benchmark and its parameters, it incurred a 90% overhead in term of server response time [35]. In our experiment, we measured the response time of Leapfrog on Apache under six scenarios, as illustrated in Table 2. The overhead of Leapfrog was about 20% even in the worst case and far below 15% in other cases.

Table 2: Response time of Apache (in milliseconds). We also indicate overheads in percentage.

	Apache+cache			**Apache**		
File Size	**Native**	**w/ PIN**	**w/ Leapfrog**	**Native**	**w/ PIN**	**w/ Leapfrog**
4K	1.017	1.019(0.22%)	1.136(11.77%)	0.98	1.118(14.0%)	1.181(20.46%)
8K	1.385	1.395(0.77%)	1.537(10.97%)	1.350	1.353(0.20%)	1.538(13.92%)
16K	2.029	2.080(2.55%)	2.169(6.90%)	1.989	1.993(0.20%)	2.161(8.66%)
512K	45.036	45.044(0.02%)	45.082(0.10%)	44.957	44.969(0.03%)	45.155(0.44%)

The performance of other patched applications was measured using the tools we developed with perl. The tool for testing `pure-ftpd` automatically logged onto the server to download the files with various sizes and sensitivity levels, and also recorded the time for accomplishing this task. This process was repeated for 1000 times in our experiment to get averaged results. We attempted to test Samba server in the similar way. A problem is that Samba requires a client to interactively enter a password in order to log in. In addition, it keeps a buffer on the client side to cache the data from prior transactions. These problems were solved in our experiment through manually cleaning the buffer and keying in the password before a timer was triggered to record the duration for serving a request. This treatment, however, limited the number of tests we could conduct. As a result, the outcomes for Samba were computed over 10 independent tests. Figure 4 illustrates the experimental results, showing that the impacts of Leapfrog on the performance of those applications were below 5% in most cases. The exception is the test of `pure-ftpd` on the small file, which introduced about 20% performance degradation. Like Pico server, this problem may also come from the small sizes of the application and the file. The performance of Napster cannot be evaluated without modifying its source code, which prevented us from measuring it in the experiment.

Overheads of the offline analysis. We also studied the performance of two major analysis steps: the taint analysis that identifies data-leaking paths and the symbolic execution that generates path conditions and TPCs. The experimental results are presented in Table 3. Performance of the instrumentation step can be hard to gauge because it may be included in the taint analysis when all the nodes host API calls and can involve manual steps when the default dummy call cannot be found from an executable's PLT (Section 6). Therefore, we did not measure it in the experiment.

As we can see from the table, the offline analysis in Leapfrog was also quite efficient: for the applications we tested, extraction of a data-leaking path took less than 13 seconds and generation of conditions costs about 1 minute. The overhead of the instruction-level taint analysis was surprisingly small because Leapfrog does not need to track the complete execution of an application. Instead, it only analyzed a data-leaking path that transfers the data from a file to the Internet. Such a path is usually short in a data-transferring application such as web server, FTP server and P2P software. For example, Apache's path for transferring medium size files contains 39615 instructions.

Table 3: Performance of the offline analysis, where apache has multiple paths.

App name	**Num of Instructions on Paths**	**Taint Analysis**	**Symbolic Execution**
Apache+cache	39558,39615,1082	2.53s,2.53s,0.02s	48.23s,50.40s,0.04s
Apache	17585,1082	0.16s,0.02s	21.53s,0.04s
pserv	386	0.006s	0.02s
Samba	445	0.09s	0.43s
Pure-ftpd	640	0.001s	0.03s
Napster	12	0.005s	0.004s
Multi-thread	236974	12.72s	0.01s

4 Discussion

The major limitation of Leapfrog is the coverage of data-leaking paths, as we discussed before. The problem can be mitigated through analyzing a binary executable using a composite of static analysis and dynamic analysis. Prior approaches which take this strategy, such as BIRD [30], are shown to work well on many real applications. Moreover, we are working on embedding into Leapfrog the techniques for exploring multiple execution paths [29], which will certainly improve coverage.

Learning of taint-propagation conditions also needs to be further studied. Our current design relies on the runtime information from the offline dynamic analysis to identify TPCs, which could miss some of them due to the specificity of the training inputs we use. This problem is very similar to the coverage issue and can therefore be mitigated through a careful selection of test inputs. It is also interesting to extend the aforementioned multi-path exploration techniques to search for the various TPCs that enable the same execution to propagate taint in different manners. This will be investigated in our future research.

The design of Leapfrog as presented in the paper has been tuned to data-transfer applications. The major privacy concern for these applications is direct transfer of sensitive documents to unauthorized remote recipients. For other applications, however, there could be many other channels for leaking information. For example, an application may choose to first save tainted data to a new file, which is labeled as public, and then manage to send it out. This threat can be effectively countered through a trivial extension of our current design to monitor all the data emitted from the application and dynamically adjust the sensitivity levels of the objects receiving them. A more challenging problem is to control the information flows within an application that allows interactions among multiple users. The problem can only be addressed through enhancing our current policy design to include compartmentalization. These issues will be studied in our future research.

The current design of Leapfrog does not consider information leaks through covert channels. For example, a program may change the content of untainted outputs according to tainted information so as to signal to an unauthorized party the sensitive information it is processing. Such an attack usually goes through the program's control flow, and therefore can be mitigated through a proper setting of taint-propagation rules, though detection of covert channels in general can be undecidable. On the other hand, the problem may not be very serious for the applications we are dealing with, as they are all legitimate.

5 Related Work

Techniques for instruction-level taint tracking have been intensively studied in these years. Numerous approaches have been proposed. Prominent examples include TaintCheck [34], TaintTrace [15], Memcheck [40], RIFLE [44] and LIFT [35]. These techniques are more powerful than Leapfrog, serving a general purpose which includes both control of information leaks and prevention of control-flow hijacking. A weakness of these approaches is performance: without hardware support, instruction-level tracking can slow down a program by an order of magnitude [34, 40]. Exceptions are TaintTrace, which introduces a mean slow-down factor of 5.5x [15], and LIFT, which has a mean slow-down factor of 3.5x [44] and causes only 6.2% degradation on Apache. However, even this performance improvement is achieved at other cost: TaintTrace reserves the entire upper half of the address space for shadow memory, which introduces a huge memory overhead; LIFT improves its performance through translating 32-bit x86 code to run on an x86-64 machine so as to use the machine's extra integer registers to record taint information, and taking 12.5% of 32-bit virtual memory as shadow memory. LIFT's excellent performance on Apache also relies on a set of optimization strategies that work extremely well on Apache. By comparison, Leapfrog is designed for attaining a limited goal: control of information leaks through data flows. Therefore, we can analyze a program at the instruction level offline to support very lightweight dataflow tracking and control online: as demonstrated in Section 3, the runtime overheads of our approach were very low for most applications we tested.

Language-based information-flow security [38] seeks to develop programming-language techniques for specifying and enforcing information flow policies. These techniques are meant to be used for building more secure programs, while Leapfrog is designed for protecting existing applications without access to their source code. Information-flow policies can also be retrofitted to legacy applications. A prominent example is taint-enhanced policy enforcement [46] which uses a source-to-source transformation to augment `C` programs with policies. This technique is designed for source code and therefore can take advantage of the optimization performed by compiler to reduce the overhead for dataflow tracking. In contrast, Leapfrog works directly on binary executables, though our current design is less versatile, lacking the ability to handle control-flow hijacking attacks.

Another technique for retrofitting legacy software has been proposed in [19, 20]. The approach is designed for adding authorization to legacy code. This objective differs from ours in that we intend to identify the source of data to support access control. In addition, that approach is dealing with source code while our technique is focused on binary executables. Other code retrofitting tools at least remotely related to our approach include CCured [31], Cyclone [23] and PrivTrans [13] that enhance the security features of `C` programs, and data-flow integrity [14] that controls the instructions from which an instruction is allowed to receive data. All these approaches only handle source code. Control-flow integrity [9] works on binary executables to control the targets a program can jump to, which is different from our objective of tracking and controlling data flows.

Some approaches attempt to mitigate information leaks without explicitly tracking data flows within a program. An example is TightLip [49], a technique that replicates the process of an application to create its "doppelganger", feeds the doppelganger with "scrubbed" data in response to the sensitive data the original process receives, and then compares the outputs of these processes to detect information leaks. This approach requires modifying operating systems to generate the doppelganger process, and faces the great challenge to synchronize the threads running in two processes. These problems are avoided by our approach.

Information flows can also be efficiently tracked and controlled at the level of system or API calls. An example is the famous Bell-LaPadula (BLP) model [10, 26] that regulates sensitive information between subjects (such as processes) and objects (such as files). This model has been implemented in SELinux [41, 17] through system-call wrapping. The same technique is also adopted by many other systems such as process coloring [22]. These approaches treat an application as a black box and do not dig into its internal information flow. As a result, they tend to be very coarse-grained and are incapable of dealing with many practical situations that need precise control. An example is a multithreaded application such as Microsoft Word that concurrently works on public and sensitive documents. BotSwat [42] attempts to improve the granularity of API-based tainted tracking through correlating the calls of which the arguments or buffers contain invariant byte sequences. This approach is fragile to the data transformation performed between calls, such as XOR-based encryption, and may introduce false positives if two calls happen to have the same arguments. Leapfrog tracks data flows according to the manner in which the application processes the data, and thus effectively addresses these problems.

At the high level, Leapfrog bears some resemblance to the technique for generating vulnerability-based signature [12] which uses taint analysis to extract an instruction sequence leading to a vulnerable buffer, and symbolic execution to identify the characteristics of the input for exploiting that buffer. However, Leapfrog is designed for solving a different problem, dataflow control, and therefore featured by several important properties for dealing with the issues that the prior approach does not need to face. First, an exploit input usually carries the control information for driving a program to the state in which the exploit can happen. This enables one to look at the input to determine whether it is malicious and how to handle it. In contrast, our approach needs to access a program's internal state to predict an execution path, as the input data usually do not carry any control information. Moreover, Leapfrog may need to track a data flow across multiple hops because the information for determining a data-leaking path may not be available when the data get into the program (see Figure 1). Second, our approach can track a data flow across the boundaries of threads to control the output of the thread that works on the sensitive data. This property is unnecessary for filtering exploit input. Third, different from the signature-generation technique which only controls the external input, Leapfrog also monitors and tracks the data flow originated from internal sources such as memory cache. Finally, our approach can regulate the way in which a program manages sensitive data through the operations such as mandatory sanitizing.

6 Conclusion and Future Work

In this paper, we present Leapfrog, a novel technique for retrofitting the executables of commodity applications with mandatory dataflow control. Our technique allows the patched applications to perform fine-grained tracking and control of sensitive data flows online at a very small overhead, which in many cases can almost be neglected. This is achieved through an offline analysis that extracts data-leaking paths and conditions for data to propagate through these paths, and a runtime control that only tracks sensitive data flows at a small set of checking nodes through verifying these conditions. We propose a security policy to protect highly sensitive data from being leaked out through unknown execution paths and design the mechanism to enforce the policy. Our technique also works on multithreaded applications and can patch an application without functionally changing its executable files. The future research will apply our technique to the software other than data-transfer applications. We also plan to improve Leapfrog for protecting multi-user applications and Windows-based software such as Microsoft Office.

References

[1] Windows Apache information leak. `http://securityvulns.com/Fnews354.html`, 2004.

[2] Systrace - Interactive Policy Generation for System Calls. `http://www.citi.umich.edu/u/provos/systrace/`, as of 2008.

[3] AppArmor Application Security for Linux. `http://www.novell.com/linux/security/apparmor/`, as of 2008.

[4] CIO Jury: IT bosses ban Google Desktop over security fears. `http://www.silicon.com/ciojury/0,3800003161,39156914,00.htm`, as of November 2007.

[5] Japanese cop puts police records on P2P network. `http://www.vnunet.com/vnunet/news/2194711/peer-peer-causes-police`, as of November 2007.

[6] Multiple vulnerabilities in pico server (pserv) v3.3. `http://www.securityfocus.com/archive/1/402045`, as of November, 2007.

[7] The objective caml system. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, as of November, 2007.

[8] ab Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.0/programs/ab.html`, as of November, 2007.

[9] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[10] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. MTR-2997, (ESD-TR-75-306), available as NTIS AD-A023 588, MITRE Corporation, 1976.

[11] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Towards automatically identifying triggerbased behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University, 2007.

[12] D. Brumley, J. Newsome, D. X. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *S&P*, pages 2–16, 2006.

[13] D. Brumley and D. X. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[14] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI*, pages 147–160, 2006.

[15] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC*, pages 749–754, 2006.

[16] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[17] J. Desai, G. Wilson, and C. Sellers. Extending selinux to meet lspp data import/export requirements. In *Security Enhanced Linux Symposium*, 2006.

[18] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (Usenix'07)*, June 2007.

[19] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *S&P*, pages 214–229, 2006.

[20] V. Ganapathy, T. Jaeger, and S. Jha. Towards automated authorization policy enforcement. In *Proceedings of Second Annual Security Enhanced Linux Symposium*, March 2006.

[21] N. S. Good and A. Krekelberg. Usability and privacy: a study of kazaa p2p file-sharing. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 137–144, 2003.

[22] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y.-M. Wang, and E. H. Spafford. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, 2006.

[23] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[24] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[25] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[26] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. *Journal of Computer Security*, 4:239–263, 1996.

[27] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy*, pages 164–178, 2007.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[29] A. Moser, C. Krügel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[30] S. Nanda, W. Li, L.-C. Lam, and T. cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *CGO*, pages 358–370. IEEE Computer Society, 2006.

[31] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[32] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, pages 65–74, 2007.

[33] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, 2007.

[34] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[35] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148, 2006.

[36] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *CGO*, pages 74–88, 2007.

[37] T. Reps and G. Rosay. Precise interprocedural chopping. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52, 1995.

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

[39] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. Plto : A link time optimizer for the intel ia32 architecture. In *Proceedings of Workshop on Binary Rewriting*, Sept 2001.

[40] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[41] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Report #01-043, NAI Labs, Dec. 2001. Revised May 2002.

[42] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *DIMVA*, pages 89–108, 2007.

[43] T. P. Team. `http://pax.grsecurity.net/`, as of November, 2007.

[44] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO*, pages 243–254. IEEE Computer Society, 2004.

[45] P. C. van Oorschot. Revisiting software protection. In *Proceedings of the Information Security Conference.*, pages 1–13, 2003.

[46] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[47] L. Xun. A linux executable editing library. `http://www.geocities.com/fasterlu/leel.htm`.

[48] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[49] A. R. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[50] S. Zdancewic. Challenges in information flow security. In R. Giacobazzi, editor, Informal proceedings of PLID'04, 2004.

Appendix

Table 4: Taint Rules in Taint Analyzer of LeapFrog.

Instruction Category	**Taint Propagation**	**Examples**
Data Movement	(1) taint is propagated to the destination if the source is tainted, (2) the taint mark of the destination operand will be cleared if the source operand is not tainted.	`mov eax,ebx`, `push eax`, `call 0x4080022`, `lea ebx, ptr [ecx+10]`
Arithmetic	(1) taint is propagated to the destination if the source is tainted, (2) there are several special cases for taint propagation. For example, in `or eax, ebx`, the taint in `ebx` will not propagate into `eax` if `eax=0xFFFFFFFF`.	`and eax, ebx`, `or ecx, ebx`, `shr`
Address Calculation	(1) an address is tainted if any element in the address calculation is tainted	`mov ebx, dword ptr [ecx+2*ebx+0x08]`
(Un)conditional Jump	no propagation in our prototype	`jmp 0x0746323`, `jnle 0x878342`, `jg 0x405687`
Others	no propagation in our prototype	`inc ecx`, `nop` `not eax`

Implementation

We implemented a prototype system of Leapfrog for evaluating its efficacy. The prototype includes an offline taint analyzer and a runtime dataflow controller, both of which were developed using Pin [28], and other tools for symbolic execution and static instrumentation. We elaborate this implementation as follows.

6.1 Taint analyzer

We implemented the taint analyzer using the Just-in-time (JIT) mode provided by Pin for an instruction-level inspection of a running program. The analyzer first identifies a data flow originated from a source (either memory source or a file) and then traces the flow instruction by instruction. An execution path is dumped out if the flow reaches an output node.

API and system calls. Our analyzer detects the external taint sources through the API or system calls that operate on file system, such as `read` and `mmap64`. A problem here is that the number of such API functions is large, and enumeration of all of them can be hard in practice[7]. Our solution is to intercept the system calls these API functions have to use. The number of such system calls is much smaller and they are also relative stable with regard to different Linux versions. Identification of these calls allows us to pinpoint the API calls that invoke them for inspecting the input data. The same approach has also been applied to locate the APIs for networking operations.

Taint propagation. The taint analyzer is implemented as a Pin tool and can directly work on the binary executables under the x86 architecture. After Pin loads an instruction trace into its code cache, our tool analyzes every instruction in the trace to extract the relation between an instruction's input and output in accordance with a set of taint-propagation rules. For example, the instruction `mov eax ebx` implies that taint can be propagated from `eax` to `ebx`. Such relations are also kept in a cache so that they can be reused when the same instruction is encountered again in the follow-up execution. Like other taint-analysis mechanisms [34, 16, 15], our analyzer uses shadow memory to maintain taint markings and taint-propagation rules. So far, it can only track data flows. This can be insufficient as taint can also go through the control flow. We plan to study this problem in the future research.

Output. Our analyzer exports all the instructions on a data-leaking path once the path has been discovered. It also specifies the locations of these instructions and tainted buffers, and the names of API calls. For example, an instruction in the Apache server is described as follows: `0x0360b8cb call 0x35fd280,,:__libc_writev`
`from /lib/tls/libc.so.6`. A problem we encountered was the size of the instruction trace, which can become huge if the analyzer dumps all the instructions on an executable path, including those within API functions. This problem was solved in our implementation with the following measures. First, all instructions belonging to an API function were filtered out, with only the name of the API left. The underneath assumption here is that the API always propagate the taint as observed during the analysis. This assumption can be released using the model of that function. Second, only the instructions within a taint-related thread were dumped. This removed the interference caused by concurrently-running threads. Third, a threshold is set to ensure the analyzer to stop at some point.

6.2 Dataflow controller

We implemented a dataflow controller using the probe mode of Pin. In this mode, Pin can be invoked at a pre-defined program location, which avoids the expensive operation for translating instructions[8].

We wrapped APIs using the probe mode, in which Pin dynamically injected our code to the beginning of an API function. The code there needs to know the location of the instruction that invokes the call, which can be achieved by checking the call stack. Implementation of mandatory sanitizing turned out to be quite challenging because of memory mapping functions such as `mmap`. These functions are widely used by high-performance severs to map the content of a file to a memory segment. The segment can be set to `read-only`, which happens in some applications such as Apache, and therefore cannot be cleaned.

[7]Even when this can be done, maintenance and retrieval of a long list of such functions can also have a significant impact on the performance of the analyzer.

[8]The probe mode in the current Pin distribution can only reliably intercept calls to library functions [28], though theoretically it can be used to dynamically instrument arbitrary program locations.

We solved this problem by intercepting a mapping call, copying the content of the segment to a new buffer, and returning the address of the buffer to the caller. This buffer can be cleaned. We also wrapped unmapping functions to ensure that the segment can be properly freed.

6.3 Other tools

The symbolic execution tool we used in our research was developed using OCamel [7]. Given an execution path, it assigns symbols to the registers and memory locations touched by the instructions on that path and emulates the execution of the path to generate symbolic expressions, as described Section 2.3.

We also developed a binary rewriting tool using a similar technique implemented in LEEL [47]. Specifically, our tool moves the ELF header and the program header table of an ELF executable file 4k bytes towards the lower end of its address space. This 4k space can be used to add in new program headers, as LEEL does. Another option our tool provides is direct placement of the new instructions to that space. This is possible because Leapfrog is designed in a way to avoid static instrumentation if possible, and when change of the original executable becomes inevitable, it just inserts a dummy call, which is no more than a few instructions. For the dynamic linking library (DLL) which starts with an address of zero, our tool appends the new space to its high-address end. To invoke the injected code, our tool replaces the instructions at a node with a `jump` to a location within the new space, moves the original instructions to that location, and appends them with a dummy call and a `jump` back to the instruction after the node.

A problem here is how to choose a dummy API call. Such a call should be quite pervasive, appearing in most executables. It must serve as a “no-op” call given right arguments so as to avoid interfering with an application’s normal operation. In our research, we chose `close(-1)` as a default dummy call, as it satisfies both conditions. However, some applications such as Apache wrap all the APIs they call with their own DLL functions, which makes the default calls no longer a feasible choice as their functions may not appear in an executable’s Procedure Linkage Table (PLT). Our solution is to let the instrumentation tool check PLT first: if the functions for default calls are not there, it just outputs all the functions in that table to let a user manually select one and set proper arguments.