Program-object Level Data Flow Analysis with Applications to Data Leakage and Contamination Forensics

Gaoyao Xiao, Jun Wang, Peng Liu, Jiang Ming, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
{gzx102, jow5222, pliu, jum310, dwu}@ist.psu.edu

ABSTRACT

We introduce a novel Data Flow Analysis (DFA) technique, called PoL-DFA (Program-object Level Data Flow Analysis), to analyze the dynamic data flows of server programs. PoL-DFA symbolically analyzes every instruction in the execution trace of a process to keep track of the data flows among program objects (*e.g.,* integers, structures, arrays), and concatenates these pieces of data flows to obtain the overall data flow graph of the execution. We leverage PoL-DFA to identify malicious data flows in data leakage and contamination forensics. In two mocked digital forensic scenarios, for data leakage and contamination respectively, we tested the ability of PoL-DFA to identify data flows among multiple inputs and outputs of server programs. Our results show that PoL-DFA can accurately determine whether the data (or the processed results) from a source file or socket flow to a certain output channel. Based on this information, security administrators can pinpoint the path of data leakage or data contamination. Different from existing dynamic DFA techniques that require excessive amount of instrumentation, PoL-DFA only requires logging the execution traces of the processes being monitored. The measured performance overhead for server programs is 4.24%, on average. The results indicate PoL-DFA is a lightweight DFA solution for data leakage and contamination forensics.

1. INTRODUCTION

We consider the following scenario: A company has a web server that hosts some web services to users on the Internet. The web page files in the web server can be updated or added through FTP by authorized employees. An employee uses FTP to upload a file that contains sensitive information to the web server (we assume the employee has the needed authority). He also uploads a modified index.html, which contains a link to that file, to overwrite the index.html file on the web server. After he goes home, he starts a web browser and downloads the file, which contains the sensitive information, from the company's web server. Later, the administrator identifies this file on the web server and wants to investigate who uploaded the file and whether the sensitive information has been leaked out. If so, the admin also wants to know which IP addresses the information has been leaked to.

We explored the literature for possible solutions. Causality relationships can be inferred from system events such as file operations, network I/O, and memory read/write, and be used for attack provenance analysis[16, 28, 18, 21]. This type of relationship is unsuitable for data flow analysis – taking the aforementioned data leakage scenario for example, even if the web server opens the file containing sensitive information and later sends a packet to the Internet, it does not necessarily indicate the packet contains the information from that file. Using system events to correlate objects and processes may result in a large number of false positive data dependencies, namely *dependence explosion*. Dynamic Taint Analysis (DTA) can trace data flows at the byte level and is free from dependence explosion. However, the high runtime overhead of DTA restricts itself as an *offline analysis* technique, *e.g.,* automatic generation of malware and exploit signatures [7, 25], and automatic configuration troubleshooting [5]. By "offline analysis", we mean the server programs being protected (*e.g.,* HTTP service), are *not* running at full speed serving a large number of online requests from clients.

In this work, we propose to tackle dependence explosion using a new fine-grained scheme named PoL-DFA (Program-object Level Data Flow Analysis). Different from DTA which is at the byte level, PoL-DFA targets at program object level. Data flows are tracked among network packets, program objects, files, and any devices modeled as file descriptors by the operating system. PoL-DFA only requires the logging of the basic blocks that the program executes. A data flow graph containing program objects, network packets, and files can then be built based on the analysis of the execution trace. While both PoL-DFA and DTA conduct instruction-by-instruction analysis, we have dramatically different design goals. DTA on binary code typically relies on dynamic binary instrumentation to propagate taint tags at the instruction level. By contrast, PoL-DFA tracks the data flow at program-object level. Our system models program objects, call stacks, and heaps of each process, and simulates the process creation operations. Compared to DTA, although PoL-DFA does not log complete runtime values, we show that our approach can still achieve a similar level of precision and better runtime performance.

CODASPY'16, March 9–11, 2016, New Orleans, LA, USA

 ISBN 978-1-4503-3935-3/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2857705.2857747

2. OVERVIEW

This section overviews the intuitions behind our approach and the system work flow.

2.1 Program-object Level Data Flow Analysis (PoL-DFA)

We define PoL-DFA (**P**rogram-**o**bject **L**evel **D**ata **F**low **A**nalysis) as *the procedure of inferring the data flow dependencies among the files and sockets ever accessed by a program through analyzing the execution trace*. We define the *execution trace* as a sequence of program statements or instructions in the order in which they have been executed at runtime. A data flow is essentially the consequence of a program execution. Involved in the execution are a series of program objects and the corresponding operations against them. The below observations show the intuitions of how PoL-DFA works.

Observation 1. The execution trace constructs, such as program statements, instructions, symbols and so forth, inherently imply data flows, even if the corresponding runtime variable values and memory addresses are unknown. The following code in Listing 1 shows a network packet being received, stored in a buffer, and then saved into a file. Knowing neither the runtime memory address of `buf` nor the runtime value of `sockfd/fd`, one can still conclude the data flow purely based on static analysis. In addition, by correlating with the system call log, one can draw an object-level data flow graph as shown in Figure 1.

Listing 1: An example of object level data flow.

```
1  buf = malloc(1024);
2  recvfrom(sockfd, buf, len, src_addr, addr_len);
3  fd = fopen('download.exe', 'w+');
4  fwrite(buf, 1024, 1, fd);
```

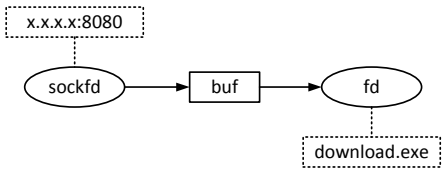


Figure 1: An object-level data flow graph resulted from the static code analysis and runtime system call log.

Observation 2. In many cases, data flow that cannot be statically determined can instead be resolved using the runtime control flow information embedded in the execution trace, including branches and loop iterations. Take the following pseudo code as an example. If the `condition` is true, the data flow will be `file A -> socket C`; otherwise, the data flow will become `file B -> socket C`. If we know which branch has indeed been taken at runtime, we can decide the correct data flow.

```
1  if (condition)
2      read in file A to buffer;
3  else
4      read in file B to buffer;
5  write buffer to socket C;
```

Observation 3. We realize that there are some cases wherein the data flow cannot be precisely derived solely based on the execution trace. The major portion falls into the situation that a variable (instead of an immediate constant) is used as an index to an array or an offset to a buffer. In such cases, one can choose to log all such concrete index/offset values at runtime. We find that by treating an array/buffer as an atomic unit, we can still correctly identify data flow in most cases (see §3.3 for further discussion).

We devise a PoL-DFA mechanism, inspired by the aforementioned three observations. In particular, we sequentially go through every instruction in the execution trace, analyze and translate the instructions into data flow propagation operations between the source operand(s) and the destination operand(s), and in the meantime maintain and populate the data flow states for the corresponding program objects (an operand typically represents an object or part of an object). Finally, a data flow starting from an input file (or socket) to an output file (or socket) can be bridged by a series of intermediate program objects. Currently, the intermediate program objects include data type instances (*e.g.,* integers, floats, pointers), data structure subfields, and arrays. Note that a file descriptor, which refers to a socket or file, is represented as an integer data type as well.

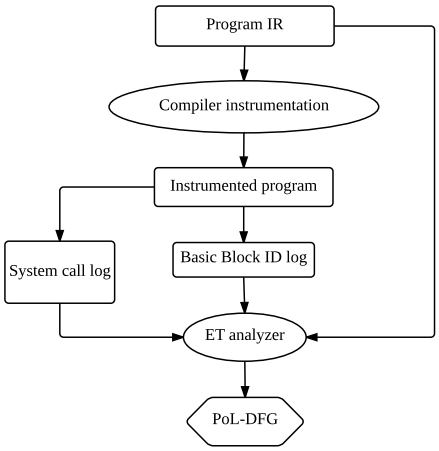


Figure 2: System work flow.

2.2 System Workflow

The workflow of our system is shown in Figure 2. The first step to deploying our system is instrumentation. We implemented a compiler extension to instrument the LLVM IR (Intermediate Representation) of the programs being monitored. The binary executables produced by our instrumentation automatically log the basic block trace of each process into files during runtime. The basic block log contains a sequence of basic block IDs, using which we can recover the execution trace of the program. The second part of the work flow is trace recovering and analysis, which is conducted by ET(Execution Trace)-Analyzer. ET-Analyzer takes basic block log, system call log (used for annotation purpose, *e.g.,* annotating the file name of a file descriptor), and LLVM IR of the program being monitored as inputs. Based on the inputs, ET-Analyzer recovers the execution trace, conducts PoL-DFA against the trace, and outputs a PoL-DFG (Program-object Level Data Flow Graph) which shows the data flows resulted from the program execution.

3. DESIGN

This section introduces the concept of LLVM IR-level execution trace, and how we infer data flows based on the trace.

3.1 LLVM IR-Level Execution Trace

For an execution of a program, the *IR-level execution trace* is a sequence of IR instructions that correspond to the native machine instructions executed. As we know, the IR of a program can be compiled into machine code, which can be executed. When the program runs, a sequence of machine instructions is executed and this sequence of machine instructions has a sequence of IR instructions as a counterpart in the form of IR. This sequence of IR instructions is called the IR-level execution trace of this execution.

The LLVM IR of a program contains a number of basic blocks. An LLVM Basic Block is simply "a container of instructions that execute sequentially" [3]. An example LLVM basic block is as below. In this LLVM Basic Block, a value `%2` is loaded through a pointer `%i`, and used as the parameter of function `%foo`. The return value `%call` is stored through a pointer `%ret`. Then the basic block exits. The execution of a program involves entering and exiting a sequence of basic blocks.

Listing 2: An example of IR basic block.

```
1 for.body:                       ; preds = %for.cond
2   %2 = load i32* %i
3   %call = call i32 @foo(i32 %2)
4   store i32 %call, i32* %ret
5   br label %for.inc
```

We statically assign a unique ID to each basic block and then at the beginning of each basic block, we insert instructions to store the ID to the buffers. We also create a thread to write the logs to hard disk at runtime. The logs show the sequence in which basic blocks are entered. With the logs, we are able to recover the IR level execution trace.

3.2 The PoL-DFA Model

Data Sources. Data sources are the files that a program reads data from and the sockets that a program receives packets from. Data sources are introduced via system calls like `read` and `recv/recvfrom/recvmsg`, as well as library functions like `fread`. Each incoming data source will be assigned a unique *tag*, which represents a unique data source. The tag will then be propagated to other program objects throughout the entire analysis whenever there is a data flow.

Data Sinks. Data sinks are the files that a program writes data to and the sockets that a program sends packets to. Tags can be propagated to data sinks via system calls such as `write`, `send/sendto/sendmsg`, and library functions like `fwrite`. A program-object level data flow graph can be built based on the tags that have arrived at each data sink. In a program-object level data flow graph, a data source could have multiple outgoing edges connecting to different data sinks; and a data sink could have multiple incoming edges starting from different data sources. Note that a file/-socket could be the data sink of one data flow, and the data source of another data flow.

Data Flow. A data flow happens when the contents of a data source are 1) directly copied and written to a data sink, or 2) computationally modified (*e.g.,* compression, encryption/decryption) before being written to a data sink. We aim to extract the data flow by looking at the operations in the execution trace and keeping track of the tag propagation among the program objects. Particularly, we build *data flow paths* between data sources and data sinks. A data flow path is a sequence of nodes through which the data flows. The start node is the data source and end node is data sink. All nodes in between are program objects involved in the operations. The operations against these program objects can be broken down into three categories: data movement, arithmetic, and neither. For a data movement operation, the tags of the source operand(s) will be propagated to the destination operand(s). For an arithmetic operation, all the tags that have arrived at any operand will be propagated to the destination.

To hold the tags, we associate each program object with an abstract metadata structure, called *object state*. For non-pointer variables, the object state contains only tags. For pointers, the object state consists of tags, the symbolic link to the program object being pointed to, and, if it belongs to a subfield of a aggregated data type (*i.e.,* struct, array, and vector), the relative offset of the target object to its "container" object. Here "container" object means the aggregate object that contains the pointer. For aggregated data types, the object state is a list of `<offset, size, tags, polPtr>` tuples, which indicates the tags associated with each subfield starting from `offset` and ending at `offset+size`. The `polPtr` stands for program-object level pointer. In case the subfield happens to be a pointer, the `polPtr` element will then be used to store a pointer object state, which links to the target object being pointed to. Table 1 summarizes the object states and the corresponding program objects.

For the lookup purpose, we build a hash map for the object states, which we call *object state map*. The object state map is a one-to-one mapping from program objects to each of their object states. When tags need to be propagated among program objects, we first look up the corresponding object states in the object state map and then copy and update the tags accordingly. We define two types of object state maps: global object state map and local object state map. There is only one global object state map throughout the entire analysis, keeping track of the object states for global variables. Local object state maps are per-function based. In other words, local object state maps are analogous to function stacks and global object state map is comparable to data segment.

3.3 Policies on Tag Propagation

In this section, we focus on tag propagation. We realize that the tags could be propagated in several possible ways. We denote the possible ways as PoL-DFA policies. We also realize that no single PoL-DFA policy is perfect. Below we present our policy based on the following principles. Principle 1—simplicity: we want everything to be as simple as possible. Principle 2—clear semantics: to avoid ambiguity. Principle 3—utility: it can handle the security officers' needs (*e.g.,* the case studies (§5.1)). We group our policy into three categories: *direct tag propagation*, *indirect tag propagation*, and *block tag propagation*. It should be noted that since the semantics of LLVM IR is at a higher level, certain types of IR instructions[1] do not have the correspondence in the assembly language. This results in the uniqueness of IR-based

[1]LLVM IR Instruction Set: http://llvm.org/docs/LangRef.html.

Table 1: Primitive data types of object state and corresponding program objects.

Object State	Elements	Corresponding Program Objects
`polVar`	`tags`	variables (*e.g.,* i16 %i)
`polPtr`	`tags, ptr, offset`	pointers (*e.g.,* i32* %ptr)
`polObj`	`<offset, size, tags, polPtr>`	arrays (*e.g.,* [4 x i8]), structures (*e.g.,* {i32, i32, i32}), and vectors (*e.g.,* <4 x i16*>)

policy over assembly-based policy. Our PoL-DFA policy is presented as follows.

I. Direct tag propagation If there is a direct data flow from source operand(s) to destination operand(s), we do direct tag propagation, that is, we make a union of the tags in the source operands' object states and copy to that of the destination. Immediate constants are considered untagged. The types of instructions involved in this category include arithmetic instructions (*e.g.,* `add`, `sub`), bit instructions (*e.g.,* `and`, `or`, `shl`), and conversion instructions (*e.g.,* `truc`, `bitcast`). Figure 3(a) shows an example of direct propagation.

There is one particular instruction `select` that needs a subtle treatment. The `select` instruction assigns either one of two source operands to the destination operand based on a condition. Due to the lack of the runtime value of the condition, we cannot determine which candidate operand to assign. In many cases one of the two source operands cannot be seen in the earlier trace because the corresponding branch has not been executed. As a result, the object state of that operand will not be found and thus we can do the tag propagation normally. If both object states can be found, we then conservatively propagate both of the two candidates' tags and give them a *Possible flag (P flag)*. Basically, we use P flags to incorporate such conservative tag propagation situations.

II. Indirect tag propagation For many instructions, there is no direct data flow among the operands. Instead, either source or destination operand is a reference (*i.e.,* pointer) to a real program object that is involved in the data flow. This type of instructions comprise a major portion of the execution trace, including `load` (reading data from memory) and `store` (writing data to memory). In such cases, we first look up the real program object (via the symbolic link kept in the object state) and then propagate tags from the real source to the real destination, based on the types of the pointer.

First, if the pointer points to a variable, we simply do the tag propagation using the object state of the variable. For example, Figure 3(b) depicts how the tag is propagated for a `load` instruction in this case. The handling of `store` instruction is similar, just with a reverse direction of tag propagation.

Second, a pointer could point to a subfield of an aggregated data type. In LLVM IR, there are mainly two categories of aggregated data types: 1) `array/vector` type which arranges element sequentially in memory, and 2) `struct` type which is a predefined collection of data members. The `GEP` (*i.e.,* `getelementptr`) instruction is normally used to return the pointer to a subfield of an aggregate data type. For the `struct` type, the `GEP` instruction will always see literal values as the offset to the subfields of a data structure. Therefore, we propagate the tag corresponds to the particular offset. For the `array/vector` type, we conservatively treat the whole array as an atomic unit and do tag propagation for the array as a whole. Figure 3(c) shows how a `GEP` involved in `struct` indexing and how a `load` in such a case is handled.

III. Block tag propagation As direct and indirect tag propagations are both for data flow between individual variables, there is another important data flow existing among buffers, *i.e.,* large pieces of contiguous memory. We also consider file and socket as such kind of buffer. So in general we call them "blocks". Instructions that can lead to data flow between blocks include system calls `read/write`, `send/recv`, and library functions `memcpy` *etc.* Generally, we treat a block as an atomic unit whenever the `size` and/or `offset` arguments are variables whose run-time values are unknown. Therefore, for the source blocks, we copy out all the tags to the destination block, including those with the P flag. For the destination blocks, we use a heuristic to decide whether to clean up existing tags:

- First, if data is stored to a block from the *beginning* (*e.g.,* reading a file `fd` to a buffer `buf`: `read(fd,buf,sz)`), we clean up all the tags in the block's object state and copy the new tag(s) into it. This policy is based on the common practice that a block is always used as an atomic buffer to *relay* data from/to a file or socket. Even if an array could be used for multiple files/sockets in different iterations, an access to it at a certain time is only meant to get its most recent content. Moreover, a `size` parameter is always used in a very careful way by programmers to enforce boundary checks in order to avoid buffer over-read bugs. Nevertheless, buffer over-read bugs do exist in reality such as the notorious Heartbleed bug [2]. Since there are better approaches addressing such vulnerabilities, here we simply assume the program is free from such over-read bugs.
- Second, if data is stored to the *middle* of a block (*e.g.,* `read(fd,buf+offset,sz)`), this indicates that the block is most likely used as a buffer to *accumulate* data from multiple data sources (*e.g.,* files concatenation). Since it is impossible to predict which part of the buffer a later-on access intends to get, we cannot simply remove all the existing tags. Instead our policy is to first set the P flag for all existing tags of the array, if any, and then merge the new tag(s) into it.

3.4 Program-object Level Data Flow Graph Generation

Our system is designed to generate two types of PoL-DFG, with program objects versus without program objects. For the PoL-DFG without program objects, it contains a set of nodes and directed edges. Each node is a system object represented by a file descriptor, *e.g.,* file, socket, standard I/O, *etc.* Each directed edge means the existence of data flow from the start node to the end node. For the PoL-DFG with program objects, a node could also be a program object. For many digital forensic needs, the details about data flow through program objects are not needed. For some

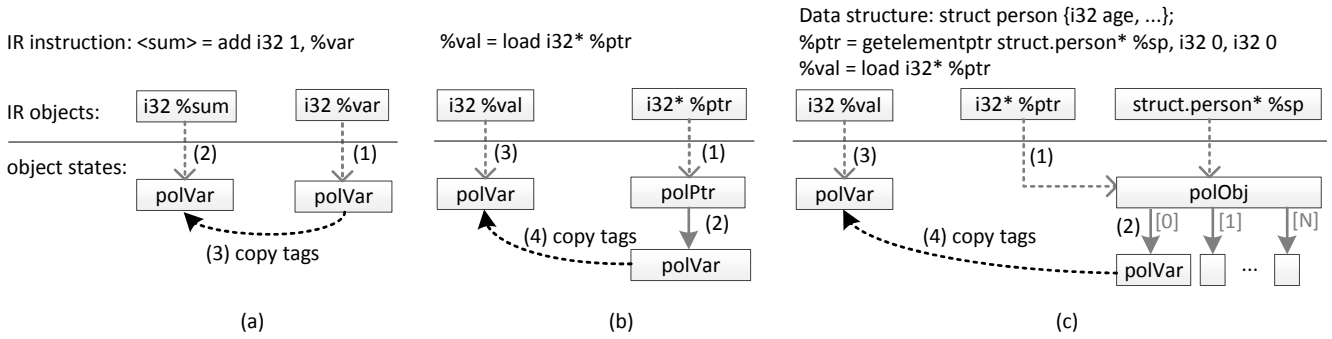


Figure 3: Tag propagation examples. (a) An example of direct tag propagation; (b) An example of indirect tag propagation; (c) An example of indirect tag propagation involving aggregated data types.

other, the officer may also want to see the data flow through program objects, so we also provide this as an option.

Node Annotation. For a node representing a file or a socket, the file name or bound IP of the node is annotated using the system call log. For one execution of a program, I/O related system calls, such as `open`, `socket`, `connect`, *etc.*, are logged in the order they are called. In the IR-level execution trace of this execution, the instructions that call these system calls (or library calls, such as `fopen`, implemented using these system calls) appear in the same order. We simply sequentially match each call instruction, which results in an I/O system call, against the corresponding entry in the system call log. The file names and IP addresses can be extracted from the arguments of system calls. Especially, a socket might be created using `socket` system call and bound with an IP address using `connect` system call. The IP address bound to this socket is extracted from the arguments of `connect`.

Edge Generation. For PoL-DFG without program objects, the edges are created when outputting system calls and library calls, such as `write`, `fwrite`, `sendmsg`, *etc.*, are found in the IR-level execution trace. An outputting system call usually write the contents in a buffer to a file or socket represented by a `fd`. If the tag list of the buffer's object state is not empty, we propagate these tags to the `fd`. For each propagated tag, an edge is created from the data source, corresponding to the tag, to the data sink represented by the `fd`. For PoL-DFG with program objects, each propagation operation of a tag results in the creation of an edge, which starts at the source node of tag propagation and ends at the destination node. The tag propagation operations include data assignment, reading from a file, writing to a file, *etc.*

4. IMPLEMENTATION

Since assembly language does not offer type abstractions (*i.e.,* integers, pointers, structure, *etc.*), we leverage LLVM [20] to base our system. The benefit is twofold: First, LLVM infrastructure enables us to conduct analysis for programs written in various programming languages (*e.g.,* C, C++, Fortran) and running on different platforms (*e.g.,* x86, PowerPC, ARM), making our tool widely applicable. Second, LLVM IR (*i.e.,* intermediate representation) has a well-defined type system and preserves high level program abstractions which can greatly ease our analysis, especially compared with low level assembly language from which the high level semantics has already been removed.

We implemented a prototype PoL-DFA system. The instrumentation component is implemented as an LLVM IR transformation pass, which has 290 LOC. The IR of programs is obtained through compiling source code using clang. The IR is instrumented and then transformed by clang to binary program. The ET-Analyzer is implemented in C++ and it has 5408 LOC. The system calls that create data sources and sinks are logged by Auditd. We use Python scripts to parse the system call logs to extract the file names and IP addresses of data sources and sinks. The PoL-DFG generation is also implemented in Python. We use a Python library named *pydot* to generate PoL-DFG based on the results of PoL-DFA. The Python scripts for parsing system call log and generating PoL-DFG have 340 LOC in total.

5. EVALUATION

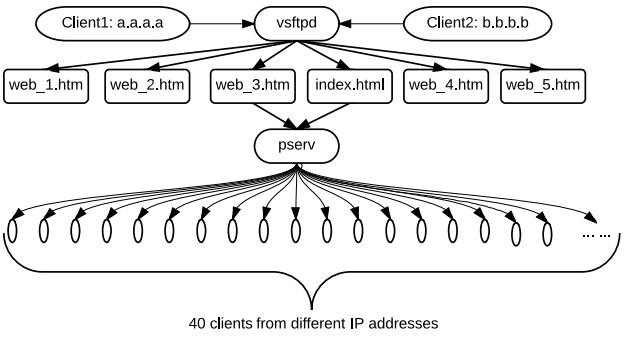
We measure the logging overhead, analysis efficiency, and space overhead, on a machine with Intel(R) Xeon(R) CPU X3440 2.53GHz, 12GB DDR3 RAM, and Samsung SSD of 250GB of model 840.

5.1 Solving Dependence Explosion

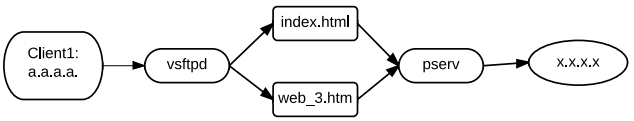
We conducted two case studies to investigate whether PoL-DFA can solve the dependence explosion problem in data leakage and contamination forensics. We consider two scenarios in which, without PoL-DFA, security administrators would have difficulty to spot malicious data flows, due to the undecidability of data flows.

Case Study 1. Information Leakage: We consider the information leakage scenario mentioned in the introduction. By conducting causality analysis, *i.e.,* inferring data dependencies based on system events, the data flow graph can reach the extent as shown in Figure 4(a). Because there is no publicly available tool for causality analysis, we infer this graph based on algorithms introduced in existing causality analysis works. The graph shows several files were sent to the web server through FTP clients on two different workstations. After the files were uploaded, the web server served requests from 40 different IP addresses. Processing these requests involved six files. Based on this graph, the admin cannot determine who uploaded the file that contains sensitive information and whether it was leaked through the 40 connections (although this file was opened by the web server, this does not mean it was sent to any client). In addition, if the admin assumes the information is leaked through the 40 connections, the graph does not tell which of the 40 is more likely to be the sink.

Figure 4(b) shows the PoL-DFG generated by ET-analyzer.

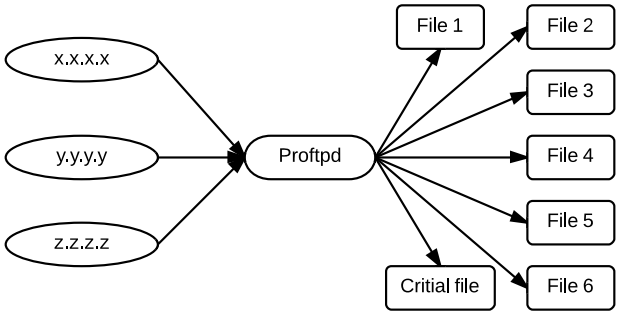


(a) Generated by course grained DFA

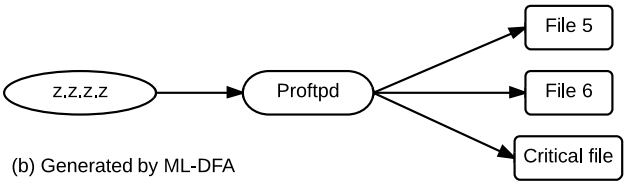


(b) Generated by ML-DFA

Figure 4: Information leakage.



(a) Generated by coarse grained DFA



(b) Generated by ML-DFA

Figure 5: Data contamination.

Table 2: Data flow graph comparison of information leakage case.

Statistics	Causality analysis	PoL-DFA
No. of nodes	50	6
No. of edges	50	6

Table 3: Data flow graph comparison of data contamination case.

Statistics	Causality analysis	PoL-DFA
No. of nodes	12	5
No. of edges	11	4

It precisely identifies that the file, which contains sensitive information, was uploaded to the web server from the FTP client of IP address *a.a.a.a*. The PoL-DFG also pinpoints that the file was indeed leaked out and the destination IP address was *x.x.x.x*. Compared with causality analysis, PoL-DFA slashes off false data flow edges and resolves the dependence explosion problem. Without false data flow edges, the admin can precisely identify who is responsible for the information leakage.

The comparison of the two graphs is shown in Table 2. The graph created by PoL-DFA has 6 nodes and 6 edges, while the other graph has 50 nodes and 50 edges. PoL-DFA narrows the scope to much fewer nodes and edges.

Case Study 2. Data Contamination: We consider a data contamination scenario as follows. A company has a file server which runs FTP service. Employees can use FTP clients on workstations to upload and download files. There are some critical files on the server. Later, the company finds that a critical file on the server was corrupted. The security administrator suspects someone overwrote the original file through FTP and wants to identify who conducted the actions. The admin constructs a data flow graph related to the critical file. Causality analysis based on system events would result in a graph as shown in Figure 5(a). There were three clients connected to the proftpd server, and seven files were uploaded to the server. Because causality analysis treats programs as black boxes, it cannot identify which IP address each file comes from. Hence the graph does not give enough information for the admin to attribute the data contamination actions to the culprit.

Figure 5(b) shows the PoL-DFG generated by ET-Analyzer. It shows that the content of the contaminated file came from packets sent from IP address *z.z.z.z*. With the PoL-DFG, the admin can determine that it was the employee from IP address *z.z.z.z* that contaminated the critical file.

The comparison of the two graphs is shown in Table 3. The graph created by PoL-DFA has 5 nodes and 4 edges, while the other graph has 12 nodes and 11 edges. The table shows that the PoL-DFG precisely pinpoints the small subset of files and sockets that are involved in the data contamination.

5.2 P-flag Edges and Accuracy

The numbers of nodes and edges for each program produced by ET-Analyzer in our case studies are shown in Table 4. Please note that some files (e.g. log file) opened or created by the server processes are not relevant to any data leakage or contamination and thus we omit these files from the table. Also, even if the data flow between two files involves multiple rounds of `read` and `write`, we use only one edge to denote this data flow.

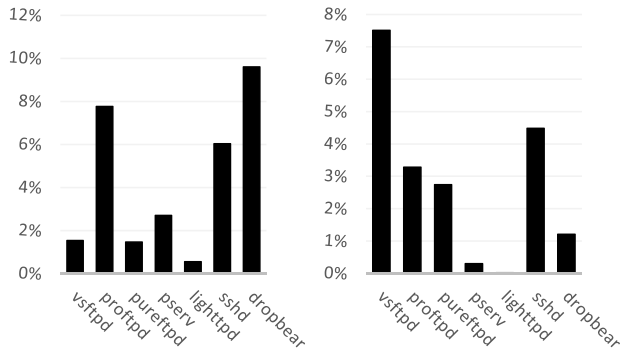
As it shows, no P-flag edge is found in our case studies. As mentioned in the description of direct tag propagation in §3.3, a P-flag edge indicates that a data flow between two program objects (or between a data source and a data sink) had possibly happened, but the ET-analyzer is not 100% sure. Having no P-flag edge means in the data flow paths from each input file (or socket) to each output file (or socket), there is no conservative propagation of tags. That is, ET-Analyzer is 100% accurate in the two case studies.

5.3 Logging Overhead

The logging overhead on server programs is measured using standard benchmark loaders if available, random inputs

Table 4: Number of P-flag Edges.

Program	# Nodes	# edges	# P-flag edges
vsftpd	4	3	0
pserv	4	3	0
pro-ftpd	5	4	0



(a) Audit on with logging execution trace to disk.

(b) Audit off with logging execution trace to disk.

Figure 6: Logging overhead on server programs.

otherwise. For lighttpd and pserv (*i.e.,* pico server), we use ApacheBench and curl-loader, respectively. For the FTP server programs, including vsftpd, proftpd, and pureftpd, we use concurrent clients to upload files of random sizes, and measure the time it takes until finishing. For sshd and dropbear server, we measure the time it takes to accept a set of randomly sized files from scp, which is a remote file copy program on Linux.

Figure 6 shows the logging overhead of the PoL-DFA system under different conditions. Figure 6a shows the logging overhead with Auditd running. Figure 6b shows the overhead measured with Auditd turned off. Figure 6 shows that in both conditions, the logging overhead is minimal. For simple programs, such as pure-ftpd, pico server, and lighttpd, the overhead is less than 5%. For more complicated programs like sshd and dropbear server, the overhead tends to be higher than 5%, but still lower than 10%. This indicates that PoL-DFA is a lightweighted DFA solution.

6. RELATED WORK

Causality Analysis based on System Events. System call logging has been widely used for building or recovering system object level dependence for the purpose of forensic analysis. It can be used to investigate attack provenance [16, 35], to track attack scope [17], and track information flows [34]. The granularity can be refined through tracking page level memory operations [18], file offsets [28], and process loops [21]. However, causality analysis is unsuitable for data flow analysis, since it cannot decisively identify existence of data flows. For example, even if a program reads from a file, it does not necessarily indicate the outputs will contain information from that file.

Dynamic information flow tracking. Dynamic information flow tracking, *a.k.a.* dynamic taint analysis (DTA) or dynamic data flow tracking (DFT), is being widely used in many domains including attack prevention [25, 31], information flow control [36, 30], data lifetime analysis [9], configuration debugging [6], malware analysis [32], and mobile security [13]. Existing dynamic information flow tracking approaches can be categorized into three categories: binary instrumentation, source code transformation, and hardware-assisted tracking. There are many optimization techniques to improve the efficiency of binary instrumentation based approaches [25, 24, 10, 15, 22, 23], and the best performance overhead reported so far [14] is about 2-3 times slowdown. Second, source code based approaches embed data flow tracking and policy checking operations into source code through source code rewriting [31, 19, 8, 33, 26, 12]. They require source code and depend on particular programming language due to their compiler-based implementation. In comparison, since ET-Analyzer is based on LLVM IR, it is language independent and does not necessarily rely on the availability of source code as there are some existing tools for transforming binary into LLVM byte code [1][4]. Hardware-assisted approaches [29, 11] can achieve very good performance, relying on specially designed hardware.

7. DISCUSSIONS AND LIMITATIONS

PoL-DFA potentially has the following limitations. First, the current implementation of PoL-DFA is vulnerable to kernel level attacks. PoL-DFA relies on Auditd to log some system calls, in order to annotate data sinks and sources, and uses system call `write` to write basic block logs to files. If the kernel is compromised, the attacker can either disable Auditd or tamper with `write` system call. However, this is not a fundamental issue. We can implement system call logging and basic block logging inside a hypervisor and get rid of this concern.

Secondly, our current implementation of logger imposes up to 10% slow down on some server programs. This overhead might be significant in efficiency-critical production systems. This issue can be eased through some optimizations. For example, we can reduce the amount of log data through logging the ID of every other IR basic block instead of logging that of every IR basic block. The IR basic block that has not been logged can be inferred based on the IR basic blocks in the context. Also, for each IR basic block, we can identify, through dynamic profiling, the most frequent successor IR basic block into which it branches. We can treat the most frequent successor as default and only need to instrument the infrequent successor IR basic block. This optimization technique is originally introduced in ShadowReplica [14]. With these techniques the runtime overhead can be further reduced.

Thirdly, it is possible that the PoL-DFG generated by our tool contains many P-flag edges, which means possible existence of data flows. Even with P-flag edges, the PoL-DFG generated by our tool still provides more knowledge to the security officer than causality analysis based on system event logging. Furthermore, while theoretically it is possible for a PoL-DFG to contain many P-flag edges, in our case studies we did not find any P-flag edge.

Finally, in this paper we don't handle control-flow-based data dependencies, *a.k.a.* implicit flows. In the field of dynamic data flow analysis such as DTA, handling implicit flows is still an open problem [27]. Our work does not aim to handle implicit flows. It aims to conduct data flow analysis at a new abstraction level.

8. CONCLUSIONS

In this paper, we present a new type of dynamic data flow analysis, called PoL-DFA, a system prototype that conducts Program-object Level DFA for the purposes of data leakage and contamination forensics. PoL-DFA is based on a new concept, straight-line execution trace enabled data flow analysis, which is a novel combination of "white-box" analysis inside a program and taint logic decoupling. PoL-DFA is designed, implemented and systematically evaluated. Experimental results show that the performance overhead is 4.24% on average for server programs, which means that PoL-DFA is a lightweight DFA solution. Our case studies show that for such security needs as data leakage and contamination forensics, PoL-DFA can solve the undecidability of data flows and help analysts spot malicious data flows.

9. ACKNOWLEDGEMENTS

This work was supported in part by NSF CNS-1223710, NSF CCF-1320605, NSF CNS-1422594, and ARO W911NF-13-1-0421 (MURI).

10. REFERENCES

[1] Dagger: decompilation to llvm ir. http://dagger.repzret.org/.

[2] The heartbleed bug. http://heartbleed.com/.

[3] Llvm basicblock class reference. http://llvm.org/docs/doxygen/html/classllvm_1_1BasicBlock.html.

[4] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *ECCS'13*.

[5] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[6] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[7] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *SP'06*.

[8] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS'08*.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX*, 2004.

[10] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07*.

[11] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*.

[12] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security'15*.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*.

[14] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. Shadowreplica: efficient parallelization of dynamic data flow tracking. In *SIGSAC'13*.

[15] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *VEE*, 2012.

[16] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP '03*.

[17] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[18] S. Krishnan, K. Z. Snow, and F. Monrose. Trail of bytes: efficient support for forensic analysis. In *CCS'10*.

[19] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC'06*.

[20] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[21] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[22] 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*.

[23] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: pipelined symbolic taint analysis. In *USENIX Security'15*.

[24] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07*.

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

[26] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: practical fine-grained decentralized information flow control. In *PLDI'09*.

[27] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *SP'10*.

[28] S. Sitaraman and S. Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *Information Assurance*, 2005.

[29] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*.

[30] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO-37*.

[31] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Usenix Security*, 2006.

[32] H. Yin, D. S. amd M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS 2007*.

[33] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SIGOPS'09*.

[34] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI'06*.

[35] H. Zhang, D. D. Yao, and N. Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *ASIACCS'14*.

[36] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *SIGOPS*.