**1libdft: Dynamic Data Flow Tracking for the Masses**

Vasileios P. Kemerlis  
Brown University  
Providence, RI, USA  
vertime@cs.brown.edu

**ABSTRACT**

Data flow tracking (DFT) deals with tagging and tracking data of interest as they propagate during program execution. DFT has been repeatedly implemented by a variety of tools for numerous purposes, including protection from zero-day attacks, detection and prevention of information leaks, and for the analysis of benign and malicious software. 1libdft is a dynamic DFT framework that unlike previous work is at once fast, reusable, and works with commodity software and hardware. In addition, it provides an API for building DFT-enabled tools that work on unmodified binaries, running on common operating systems and hardware, thus facilitating research and rapid prototyping. 1libdft is available as open source software. During the past ~10 years, the research community has used, or extended, our research prototype to facilitate a plethora of tasks that are related to security and privacy topics, leading to numerous publications with meaningful impact.

**KEYWORDS**

Data flow tracking, information flow tracking, taint analysis

**1 INTRODUCTION**

Data-flow tracking (DFT) [8], also known as information-flow tracking (IFT) [45] or taint tracking/analysis [3], is a well-established technique that deals with the tagging and tracking of “interesting” (i.e., selected) data as they propagate during program execution.

DFT has many uses, such as analyzing malware behavior [16, 40], hardening software against zero-day attacks [6, 19, 27, 31, 37, 41, 42], detecting and preventing information leaks [17, 39, 52, 54], debugging software misconfigurations [4], fuzz testing [22, 43], performing forensic investigation [25] and data provenance [44], as well as facilitating exploitation [28, 38, 49]. From an architectural perspective, it has been integrated into full system emulators [11, 12, 20, 40] and virtual machine monitors [5, 17, 21, 36], retargeted into unmodified binaries using dynamic binary instrumentation [18, 27, 34, 42], and added to source/binary codebases using source-to-source/binary code transformations [8, 51]. Proposals have also been made to implement it in hardware [13, 45, 47, 50], but they have had little appeal to hardware vendors.

1libdft argues that a practical DFT implementation should be concurrently (i) fast, (ii) reusable, and (iii) applicable to commodity hardware and software. 1libdft was originally published at the ACM VEE 2012, and released as open source software [26]. Our prototype, distributed in the form of a shared library, implements dynamic DFT using Intel’s Pin dynamic binary instrumentation framework [33], and its performance is comparable or better than that of previous work, incurring slowdowns that range between 14% – 6x.

In addition, it is versatile and reusable by providing an extensive API that can be used to implement DFT-powered tools. Finally, it runs on commodity systems. During the past ~10 years, the research community has used, or extended, 1libdft to facilitate a plethora of tasks that are related to security and privacy topics, leading to numerous publications with meaningful impact.

**2 OVERVIEW**

**2.1 Data Flow Tracking**

1libdft defines DFT as “the process of accurately tracking the flow of selected data throughout the execution of a program” [27]. This process is characterized by the following three aspects.

1. **Data sources.** Data sources are program or memory locations, where data of interest “enter” the respective (i.e., traced) process, usually after the execution of a function or system call. Data originating from these sources are tagged. For instance, if we define files as a source, the read call in Figure 1 would result in tagging data.

2. **Data tracking.** During program execution, tagged data are tracked as they are copied and altered by program instructions. Consider the code snippet in Figure 1, where data has already been tagged in ln. 3. The while loop that follows (ln. 4–5) calculates a checksum and stores the result in csum, effectively creating a data-flow dependency between csum and data.

3. **Data sinks.** Data sinks are also program or memory locations, where one can check for the existence of tagged data, usually for inspecting or enforcing data flows. For instance, tagged data may not be allowed in certain memory areas or function arguments. Consider again the code snippet in Figure 1, where in ln. 7 csum is written to a file. If files are defined as data sinks, the use of write with csum may trigger a user-defined action.

**Dynamic vs. static DFT.** Performing DFT requires additional memory for storing data tags, while the program itself needs to be extended with tag propagation logic, and data tagging and checking logic, at the sources and sinks respectively. The additional code for these tasks is frequently referred to as instrumentation code, and can be injected either statically (e.g., during source code development or at compile/link time), or dynamically using virtualization or dynamic binary instrumentation (DBI).

![Figure 1: Example of code with data dependencies.](image-url)
2.2 Design and Implementation

libdft is designed for use with the Pin DBI framework [33] to facilitate the creation of Pintools that employ dynamic DFT.

Briefly, Pin consists of a virtual machine (VM) library, and an injector that attaches the VM in already running processes or new ones that launches itself. Pintools are shared libraries that leverage Pin’s extensive API to inspect and modify a binary at the instruction level (dynamically, at runtime). libdft is also a library, which can be used by Pintools to transparently apply fine-grained DFT on binaries running over Pin. Importantly, it provides its own API (§2.3) that enables tool authors to customize libdft by specifying data sources and sinks, or modify the tag propagation policy.

When a user attaches to an already running process, or launches a new one using a libdft-enabled Pintool, the injector first loads Pin’s runtime and then passes control to the tool. There are three types of locations that a libdft-enabled tool can use as a data source or sink: (1) program instructions; (2) function calls, and (3) system calls. It can “tap” these locations by installing callbacks that get invoked when a certain instruction is encountered, or when a certain function or system call is made. These user-defined callbacks drive the DFT process by tagging or un-tagging data, and monitoring or enforcing data flow. Figure 2 sketches the memory image of a process running under a libdft-enabled Pintool. The highlighted boxes mark the locations where the tool author can install callbacks. For instance, the user can tag the contents of the buffer returned by the read system call (as in the examples shown in Figure 1) or check whether the operands of indirect call instructions are tagged (e.g., the eax register in Figure 2).

Data Tags. libdft stores data tags in a tagmap (Figure 3), which contains a process-wide data structure (mem_bitmap) or STAB tseg in Figure 3; shadow memory) for holding the tags of data stored in memory and a thread-specific structure that keeps tags for data residing in CPU registers (vcpu in Figure 3). The format of the tags stored in the tagmap is determined by two factors: (a) the granularity of the tagging, and (b) the size of the tags.
Tagging granularity: libdft uses byte-level tagging granularity, since a byte is the smallest addressable chunk of memory in most architectures, including x86 (i.e., the target platform of libdft). This choice allows fine-grained tracking for most practical purposes and strikes a balance between usability and performance [40].

Tag size: libdft offers two different tag sizes: (i) byte tags for associating up to 8 distinct values or colors to each tagged byte (every bit represents a different tag class), and (ii) single-bit tags (i.e., data are either tagged or not). The first allows for more sophisticated tracking and analysis tools, while the second enables tools that only need binary tags for conserving memory.

Tag Propagation. Tag propagation is accomplished using Pin’s API to both instrument and analyze the target process. In Pin’s terms, instrumentation refers to the task of inspecting the instruction stream of a program for determining what analysis routines should be inserted where. For instance, libdft inspects every program instruction that (loosely stated) moves or combines data to determine data dependencies. Due to the complexity and inherent redundancy of the x86 ISA, the instrumentation engine of libdft (see Figure 3) consists of ~3000 lines of code (LOC) in C++.

On the other hand, analysis refers to the actual routines, or code, being retrofitted to execute before, after, or instead of the original code. libdft injects analysis code for implementing the tag propagation logic, based on the data dependencies observed during instrumentation. Figure 4 shows an excerpt from different types of analysis routines in the case of bit-sized tags. The analysis routines of libdft (see Figure 3) are made up of ~2500 C LOC, and include only arithmetic, logical, and memory operations to ensure that Pin will inline the analysis code into the target application’s code (i.e., to minimize the runtime slowdown incurred by DFT).

The original (i.e., application) code and libdft’s analysis routines are translated by Pin’s just-in-time (JIT) compiler for generating the (final) code that will actually run. This occurs immediately before executing an application’s code sequence for the first time, and the result is placed in a code cache (also depicted in Figure 2), so as to avoid repeating this process for the same code sequence in the future. Our injected (i.e., analysis) code executes before application instructions, tracking data as they are copied between registers, and between registers and memory, thus achieving fine-grained DFT. Pin’s VM ensures that the target process runs entirely from within the code cache by interpreting all instructions that cannot be executed safely otherwise (e.g., indirect branches). Moreover, a series of optimizations such as trace linking and register re-allocation are applied for improving performance [33].

Finally, libdft allows tools to modify the default tag propagation policy, by registering their own instrumentation callbacks via its API, for instructions of interest. This way tool authors can tailor the data tagging according to their needs, cancel tag propagation in certain cases, or track otherwise unhandled instructions.

Fast Dynamic DFT. To keep libdft’s overhead low, we carefully examined how DBI frameworks (such as Pin) operate, and identified a set of development practices that should be avoided. Pin’s overhead primarily depends on the size of the analysis code injected, but it can frequently be higher than anticipated due to the structure of the analysis code itself. Specifically, the registers provided by the underlying architecture will be used to execute both application code, as well as code that implements the DFT logic.

This will force the DBI framework to spill registers (i.e., save their contents to memory and later restore them), whenever an analysis routine needs to utilize registers already allocated. Therefore, the more complex the code, the more registers have to be spilled.

Additionally, certain types of instructions must be avoided due to certain side-effects. For instance, spilling the eflags register in the x86 architecture is expensive in terms of processing cycles, and is performed by specialized instructions (pushf, pushfd). As a result, including instructions in analysis code that modify this register should be done sparingly. More importantly, test-and-branch operations have to be avoided altogether, since they result into non-inlined code. In particular, whenever a branch instruction is included in the DFT code, Pin’s JIT engine will emit a function call to the corresponding analysis routine, rather than inline the code of the routine along with the instructions of the application.

Imposing such limitations on the implementation of any dynamic DFT tool is a challenge. Our implementation takes into consideration these issues, in conjunction with Pin, to achieve good performance. More specifically, we observed that the number of instructions, excluding all types of jumps, which Pin can inline is ~20. Hence, we introduce two guidelines for the development of efficient tag propagation code: (1) tag propagation should be branch-less, and (2) tagmap updates should be performed with a single assignment.

Both of them serve the purpose of aiding the JIT process to inline the injected code and minimize register spilling. Moreover, we force Pin to use the fastcall x86 calling convention, for making the DFT code faster and smaller, while we also implement four optimizations (i.e., fast_vcpu, fastrep, huge_tlb, and tmap_col) to further minimize the runtime and memory overhead(s) incurred by libdft. (Interested readers are referred to our VEE 2012 publication for a detailed description of the above [27].)

The design of libdft provides the foundation for a framework that satisfies all three properties listed in Section 1, while by taking into consideration the limitations discussed above, we achieve low overhead. Moreover, the extensive API of libdft makes it reusable, as it enables users to customize it for use in various domains, such as security, privacy, program analysis, and debugging. Finally, the last property is satisfied through the use of a mature, rather than an experimental and feature-limited, DBI platform for providing the apparatus to realize DFT for a variety of popular systems (e.g., x86 and x86-64 [2] Linux and Windows [15] OSes).

2.3 libdft-powered Tools

One of the most frequent incarnations of DFT has been that of dynamic taint analysis (DTA). DTA operates by tagging all data coming from the network, filesystem, etc., as tainted, tracking their propagation, and alerting the user when they are used in a way that could compromise program integrity.

In this case, the network is the source of “interesting” data, while instructions that are used to control a program’s flow are the sinks. For the x86 architecture, these are jumps and function calls with non-immediate operands, as well as function returns. Attackers can manipulate the operands of such instructions, by exploiting various types of software memory errors, such as buffer overflows and format string vulnerabilities.
They can then seize control of the program by redirecting execution to existing code (e.g., return-to-libc, ROP [49]), or their own injected instructions. In this section, we describe the design and implementation of a DTA tool, namely libdft-DTA, which we implemented in approximately 450 LOC in C++, using libdft with bit-sized tags and the API calls shown in Table 1.

We only list part of the API used for the development of the tool, due to space considerations. First, libdft-DTA invokes libdft-\_init() for initializing libdft and allocating the tagmap. Next, it uses syscall_set_post() for registering a set of system call hooks to pinpoint untrusted data. Specifically, it monitors the socket API (i.e., socket and accept) for identifying PF_INET, PF_INET6 socket descriptors. It also hooks the dup, dup2, and fcntl system calls to ensure that duplicates of these descriptors are also tracked. Each time a system call of the read or recv family is invoked with a monitored descriptor as argument, the memory locations that store the network data are asserted using tagmap_setn(). libdft-DTA checks if tainted data are used in indirect control transfers (i.e., loaded on eip) using ins_set_post() with ret, jmp, and call instructions. In particular, it instruments them with a small code snippet that returns the tag markings of the instruction operands and target address (i.e., branch target). If any of the two is tainted, execution halts with an informative message containing the offending instruction and the contents of eip. In addition, for protecting against attacks that alter system call arguments, libdft-DTA also monitors the exece system call for tainted parameters.

### 3 IMPACT

libdft was originally published at the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), while, upon its publication, the authors made the accompanying software artifact publicly-available as open source software.

Following its release, libdft was used in-house to study novel methods and techniques for cross-process and cross-host taint tracking (Taint-Exchange [53]), data auditing in cloud settings (Cloudopsy [52], CloudFence [39]), adaptive hardening (VAP [19]), as well as low-overhead DFT via means of offline data-flow analysis and parallelization (TFA [24], ShadowReplica [23]). The names in parentheses correspond to prototypes of systems that build upon the open-source codebase of libdft.

In addition to the above, the research community has endorsed libdft, and used or extended our research prototype to facilitate a plethora of tasks that are related to security and privacy topics. During the past ∼10 years, libdft has lead to multiple subsequent publications with meaningful impact. In what follows, we indicatively mention a couple of such works, which demonstrate the versatility of our framework, and its ability to facilitate different security and/or privacy tasks.

In particular, the research community has used libdft as the foundation of tools and frameworks for fuzz testing (VUzzzer [43], TIFF [22]), runtime error repair and containment (RCV [31]), malware dissection (BluePill [16]), forensic investigation (RAIN [25]), system-wide IFT (SHRIFT [32]), data provenance (DataTracker [44]), as well as context-sensitive control-flow integrity (FCCFI [41]).

Furthermore, libdft has also been used for building tools that support offensive research, like the discovery of code-reuse gadgets (Newton [49]), identification of primitives for bypassing information-hiding-based isolation (MAPScanner [38]), reverse-engineering of custom memory allocation routines (MemBrush [9, 10]), and the discovery of crash-resistant exploitation primitives [28].

In contrast to the above, many studies also use libdft as a standard benchmark in terms of DFT performance and/or effectiveness. FlowWalker [14], Phosphor [5], TaintPipe [35], LDX [29], DECAF [20], StraightTaint [34], TaintInduce [12], LATCH [47], Taint Rabbit [18], SelectiveTaint [8], and PolyCruise [30] are all representative works that compare against libdft to demonstrate that they have indeed advanced the state-of-the-art in DFT.

Interestingly, libdft has also been used for plagiarism detection (DYIKS-PD [46]), while BMW has leveraged it for building a security architecture that safely allows the use of third-party applications in automotive settings [7]. Lastly, and most importantly, libdft is used in “Practical Binary Analysis” [1], a recent book on reverse engineering, as a teaching apparatus, while ports to the x86-64 architecture and the Windows platform have also been successfully demonstrated [2, 15].

### 4 AVAILABILITY

libdft is available as open source software (under a modified BSD license) at: https://github.com/brown-ssl/libdft/

### REFERENCES


