libdft: Practical Dynamic Data Flow Tracking for Commodity Systems

Vasileios P. Kemerlis Georgios Portokalidis Kangkook Jee Angelos D. Keromytis

Network Security Lab
Department of Computer Science
Columbia University, New York, NY, USA
{vpk, porto, jikk, angelos}@cs.columbia.edu

Abstract

Dynamic 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 and cross-site scripting attacks, detection and prevention of information leaks, and for the analysis of legitimate and malicious software. We present libdft, a dynamic DFT framework that unlike previous work is at once *fast*, *reusable*, and works with *commodity software and hardware*. libdft 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.

We explore different approaches for implementing the low-level aspects of instruction-level data tracking, introduce a more efficient and 64-bit capable shadow memory, and identify (and avoid) the common pitfalls responsible for the excessive performance overhead of previous studies. We evaluate libdft using real applications with large codebases like the Apache and MySQL servers, and the Firefox web browser. We also use a series of benchmarks and utilities to compare libdft with similar systems. Our results indicate that it performs at least as fast, if not faster, than previous solutions, and to the best of our knowledge, we are the first to evaluate the performance overhead of a fast dynamic DFT implementation in such depth. Finally, libdft is freely available as open source software.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Monitors; D.2.13 [*Reusable Software*]: Reusable libraries

General Terms Design, Performance, Security

Keywords Data flow tracking, dynamic binary instrumentation, taint analysis, information leak detection, exploit prevention

VEE'12, March 3–4, 2012, London, England, UK.

1. Introduction

Dynamic data flow tracking (DFT), also referred to as information flow tracking, is a well known technique that deals with the tagging and tracking of "interesting" data as they propagate during program execution. DFT has many uses, such as analyzing malware behavior [21], hardening software against zero-day attacks (*e.g.,* buffer overflow, format string) [2, 18, 22], detecting and preventing information leaks [10, 31], and even debugging software misconfigurations [1]. From an architectural perspective, it has been integrated into full system emulators [5, 21] and virtual machine monitors [13, 17], retrofitted into unmodified binaries using dynamic binary instrumentation [22], and added to source codebases using source-to-source code transformations [28]. Proposals have also been made to implement it in hardware [8, 24, 26], but they had little appeal to hardware vendors.

Previous studies utilized DFT to investigate the applicability of the technique into a particular domain of interest, producing their own problem-specific and ad hoc implementations of software-based DFT that all suffer from one or more of the following issues: *high overhead*, *little reusability* (*i.e.,* they are problem specific), and *limited applicability* (*i.e.,* they are not readily applicable to existing commodity software). For instance, LIFT [22] and Minemu [2] use DFT to detect security attacks. While fast, they do not support multithreaded applications (the first by design). LIFT only works with 64-bit binaries, while Minemu only with 32-bit binaries, featuring a design that requires extensive modifications to support 64-bit architectures. More importantly, they focus on a single problem domain and cannot be easily modified for use in others.

More flexible and customizable implementations of fine-grained DFT have also failed to provide the research community with a practical and reusable DFT framework. For example, Dytan [6] focuses on presenting a configurable DFT tool that supports both data and control flow dependencies. Unfortunately, its versatility comes at a high price, even when running small programs with data flow dependencies alone (control flow dependencies further impact performance). For instance, Dytan reported a 30x slowdown when compressing with `gzip`, while LIFT reports less than 10x. Although the experiments may not be directly comparable, the significant disparity in performance suggests that the design of Dytan is not geared towards low overhead.

This paper argues that a practical dynamic DFT implementation needs to address all three problems listed above, and thus it should be concurrently *fast*, *reusable*, and *applicable to commodity hardware and software*. We introduce libdft, a meta-tool in the form of a shared library that implements dynamic DFT using Intel's Pin dynamic binary instrumentation framework [16]. libdft's performance is comparable or better than previous work, incurring

slowdowns that range between 1.14x and 6.03x for command-line utilities, while it can also run large server applications like Apache and MySQL with an overhead ranging between 1.25x and 4.83x. 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. Our current implementation works with x86 binaries on Linux, while we plan to extend it to run on 64-bit architectures and the Windows operating system (OS). libdft introduces an efficient, 64-bit capable, shadow memory, which represented one of the most serious limitations of earlier works, as flat shadow memory structures imposed unmanageable memory space overheads on 64-bit systems, and dynamically managed structures introduce high performance penalties. More importantly, libdft supports multiprocess and multithreaded applications, by trading off memory for assurance against race conditions (see Section 7), and it does not require modifications to programs or the underlying OS.

The contributions of this paper can be summarized as follows:

- We discuss the design and implementation of a fast and reusable shared DFT library for commodity software. Specifically, we investigate and identify the underlying reasons responsible for the performance degradation incurred by previous DFT tools and present a design that minimizes it. We approach the problem from a systems perspective and attempt to answer the following questions: What are the performance boundaries of such a DFT tool? What practices need to be avoided by practitioners and system implementors? What is the source of the overhead? We also present a set of novel optimizations for further improving the performance of DFT.
- We evaluate the performance of libdft using real applications that include complex and large software such as the Apache and MySQL servers, and the Firefox web browser. libdft achieves performance similar, or better, than previous work, while being applicable to a broader set of software. Moreover, our extensive evaluation establishes a set of bounds regarding the performance of DFT. To the best of our knowledge, we are the first to perform such an extensive evaluation of a DFT framework.
- We present the development of a libdft-powered tool, namely libdft-DTA, to demonstrate the reusability of libdft as well as its capabilities. libdft-DTA performs dynamic taint analysis (DTA) to detect zero-day attacks similarly to TaintCheck [18], Eudaemon [20], and LIFT [22]. We show that our versatile API can be used for developing an otherwise complex tool in approximately 450 lines of C++ code.
- Our implementation is freely available as a shared library and can be used for developing tools that transparently (*i.e.,* without requiring any change on applications or the underlying OS) make use of DFT services. Developers can use the API provided to easily define data of interest, and then capture their use at arbitrary points. In this way, libdft facilitates research and rapid prototyping, by allowing potential users to focus on solving a particular problem (*e.g.,* detecting information leaks or application misconfigurations), rather than dealing with the elaborate details of an information flow tracking framework.

The remainder of this paper is organized as follows: Section 2 introduces DFT and discusses the differences between dynamic and static DFT approaches. We present libdft in Section 3 and elaborate on its implementation in Section 4. Section 5 discusses the use of libdft in various tools and presents its API through the creation of a DTA tool, which we evaluate along with libdft in Section 6. Section 7 examines the limitations of the current implementation, along with future considerations. Related work is presented in Section 8 and conclusions are in Section 9.

```
1: unsigned char csum = 0;
2:
3: bcount = read(fd, data, 1024);
4: while(bcount-- > 0)
5:       csum ^= *data++;
6:
7:  write(fd, &csum, 1);
```

(a) Data flow dependency

```
1: int authorized = 0;
2:
3: bcount = read(fd, pass, 12);
4: MD5(pass, 12, phash);
5: if (strcmp(phash, stored_hash) == 0)
6:     authorized = 1;
7: return authorized;
```

(b) Control flow dependecy

Figure 1. Examples of code with data dependencies.

2. Data Flow Tracking

DFT has been a popular subject of research, primarily employed for enforcing safe information flow and identifying illegal data usage. In past work, it is frequently referred to as information flow tracking (IFT) [24] or taint analysis. This work defines DFT as: "the process of *accurately* tracking the flow of *selected* data throughout the execution of a program or system". This process is characterized by three aspects, which we will attempt to clarify with the help of the code-snippets shown in Figure 1.

Data sources Data sources are program or memory locations, where data of interest enter the system, usually after the execution of a function or system call. Data coming from these sources are tagged and tracked. For instance, if we define files as a source, the `read` call in Figure 1 would result in tagging `data` and `pass`.

Data tracking During program execution, tagged data are tracked as they are copied and altered by program instructions. Consider code snippet (a) in Figure 1, where `data` has already been tagged in line 3. The while loop that follows calculates a simple checksum (XOR all the bytes in `data`) and stores the result in `csum`. In this case, there is a data flow dependency between `csum` and `data`, since the former directly depends on the latter. On the other hand, `authorized` in (b) is indirectly affected by the value of `phash`, which in turn depends on `pass`. This is frequently called a control flow dependency, and in this work, we do not consider cases of implicit data flow that are in accordance with previous work on the subject [18, 24]. Dytan made provisions for conditionally handling such control-flow dependencies, but concluded that, while useful in certain domains, they frequently lead to an explosion in the amount of tagged data and to incorrect data dependencies [6]. Ongoing work seeks to address these issues [15].

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

Dynamic versus static DFT Performing DFT requires additional memory for the data tags. Also, 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 that is frequently referred to as instrumentation code, and can be injected either statically (*e.g.,* during source code development and at compile/loading time), or dynamically using virtualization or dynamic binary instrumentation (DBI). Static systems apply DFT by recompiling software using a modified compiler [25], or a source-to-source transformation engine [28]. Conversely, the dynamic ones can be directly applied on unmodified binaries, including commercial off-the-shelf software [22, 27, 31]. In both cases, software needs to be extensively instrumented for associating data with some kind of tag and injecting logic that asserts tags at the sources, propagates them according to the data dependencies defined by the program semantics, and finally, inspecting the sinks for the presence of tagged data. Dynamic solutions, albeit being much slower than static ones, have the advantage of being immediately, and incrementally, applicable to already deployed software.

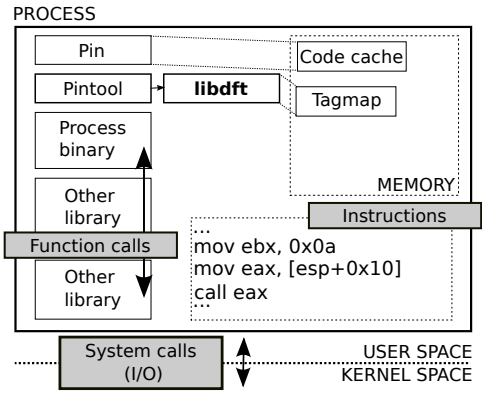


Figure 2. Process image of a binary running under libdft. The highlighted boxes describe possible data sources and sinks that can be used with libdft.

3. Design

We designed libdft for use with the Pin DBI framework 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 processes that launches itself. Pintools are shared libraries that employ Pin's extensive API to inspect and modify a binary at the instruction level. libdft is also a library, which can be used by Pintools to transparently apply fine-grained DFT on binaries running over Pin. More importantly, it provides its own API (presented in Section 5) that enables tool authors to easily customize libdft by specifying data sources and sinks, or even 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 injects 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: *program instructions*, *function calls*, and *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) and check whether the operands of indirect `call` instructions are tagged (*e.g.,* the `eax` register in Figure 2).

3.1 Data Tags

libdft stores data tags in a tagmap, which contains a process-wide data structure (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. The format of the tags stored in the tagmap is determined by mainly two factors: (a) the granularity of the tagging, and (b) the size of the tags.

Tagging granularity In principle, we could tag data units as small as a single bit, or as larger contiguous chunks of memory. The former enables us to perform very fine-grained and accurate DFT, while using larger granularity means the data tracking will be coarser and more error prone. For instance, with page-level granularity, moving a single byte (tagged) into an untagged location will result into tagging the whole page that contains the destination, thus "polluting" adjacent data. However, choosing extremely fine-grained tagging comes at a significant cost, as more memory space is needed for storing the tags (*e.g.,* using bit-level tagging, 8 tags are necessary for a single byte and 32 for a 32-bit register). More importantly, the tag propagation logic becomes complicated, since data dependencies are also more intricate (*e.g.,* consider adding two 32-bit numbers that only have some of their bits tagged). libdft uses byte-level tagging granularity, since a byte is the smallest addressable chunk of memory in most architectures. Our choice allows us to offer sufficiently fine-grained tracking for most practical purposes and we believe that it strikes a balance between usability and performance [21].

Tag size Orthogonally to tagging granularity, larger tags are more versatile as they allow for different types of data to be tagged uniquely (*e.g.,* each byte could be tagged using a unique 32-bit number). Unfortunately, larger tags require complex propagation logic and more storage space. libdft offers two different tag sizes: (a) byte tags for associating up to 8 distinct values or *colors* to each tagged byte (every bit represents a different tag class), and (b) 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.

3.2 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 binary instructions 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. On the other hand, analysis refers to the actual routines, or code, being retrofitted to execute before, after, or instead of the original code. In our case, we inject analysis code implementing the tag propagation logic, based on the data dependencies observed during instrumentation.

The original code and libdft's analysis routines are translated by Pin's just-in-time (JIT) compiler for generating the code that will actually run. This occurs immediately before executing a 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. Our injected 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 [16].

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, and cancel tag propagation in certain cases or track otherwise unhandled instructions.

3.3 Challenges for Fast Dynamic DFT

To keep libdft's overhead low, we carefully examined how DBI frameworks, such as Pin, work for identifying the 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 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, thus forcing 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.

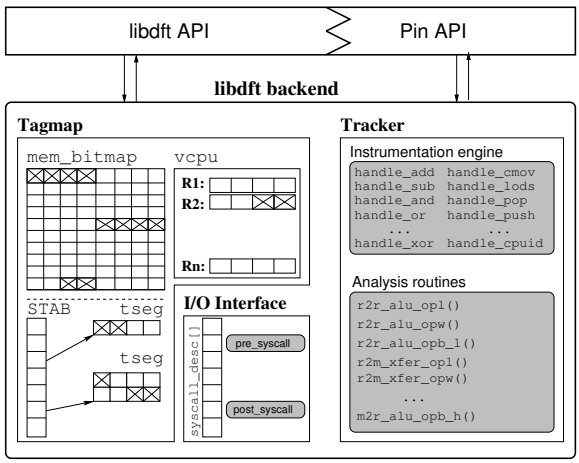


Figure 3. The architecture of libdft. The shaded components of I/O interface and tracker illustrate the instrumentation and analysis code that implements the DFT logic, whereas the x-marked regions on the tagmap indicate tagged bytes.

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.

The design of libdft provides the foundation for a framework that satisfies all three properties listed in Section 1. 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 Linux and Windows OSs).

4. Implementation

We implemented libdft using Pin 2.9, which is currently available for Linux and Windows OSs. Our prototype works with unmodified multiprocess and multithreaded applications running on 32-bit x86 CPUs over Linux, but can be extended with moderate effort to operate on the x86-64 architecture and the Windows OS (we discuss future ports in Section 7). The main components of libdft are illustrated in Figure 3.

4.1 The Tagmap

The implementation of the *tagmap* plays a crucial role in the overall performance of libdft, since the injected DFT logic constantly operates on data tags.

4.1.1 Register Tags

We store the tags for all 8 general purpose registers (GPRs) of the x86 architecture in the `vcpu` structure, which is part of the tagmap (see Figure 3). Note that we tag and track only the registers that can be directly used by applications (like the GPRs). Registers such as the instruction pointer (`EIP`), `EFLAGS`, and segment registers, are not tagged or traced. Recall that according to our definition of DFT, we only track direct data flow dependencies, so it is safe to ignore `EFLAGS`. However, instructions that are executed conditionally, based on the contents of `EFLAGS`, are handled appropriately (*e.g.,* `CMOVcc`, `SETcc`). Moreover, floating point registers (`FPU`), as well as SSE registers (`XMM`, `MMX`), are currently ignored for the sake of simplicity. To support these registers in the future, we only need to enlarge `vcpu`.

The tagmap holds multiple `vcpu` structures, one for every thread of execution. Specifically, libdft captures the thread creation and thread termination events of an application and dynamically manages the number of `vcpu` structures. We locate the appropriate structure for each thread using its virtual id (*i.e.,* an incremental value starting from zero) that is assigned by Pin to every thread. In the case of bit-sized tags, we use one byte to hold the four 1-bit tags needed for every 32-bit GPR, so that the space overhead of `vcpu` is 8 bytes for each thread. Similarly, in case of byte-sized tags, we need 4 bytes for every 32-bit GPR, and hence the space overhead of `vcpu` becomes 32 bytes per thread.

4.1.2 Memory Tags

Bit-sized tags When libdft is configured to use bit-sized tags, it stores memory tags in a flat, fixed-size structure (see `mem_bitmap` in Figure 3) that holds one bit for each byte of process addressable memory. The total size of the virtual address space in x86 systems is 4GB (2^{32}), however the OS reserves part of that space for itself (*i.e.,* the kernel). The amount of space reserved for the kernel depends on the OS, with Linux usually adopting a 3G/1G memory split that leaves 3GB of address space for processes. In this case, we require 384MB to be contiguously reserved for the tagmap.

The memory tags of address $vaddr$ can be obtained as follows: `tval = mem_bitmap[vaddr >> 3] & (MASK << (vaddr & 0x7))`. Specifically, we use the 29 most significant bits (MSBs) of $vaddr$, as byte index in `mem_bitmap`, for selecting a byte that contains the tags of $vaddr$. Then, we treat the 3 least significant bits of $vaddr$ as bit offset within the previously acquired byte, and by setting `MASK` to 0x1 we obtain the tag bit for a single byte. Similarly, if `MASK` is 0x3, or 0xF, we obtain the tag bits for a word, or double word, respectively. The address space overhead imposed by `mem_bitmap` is 12.5%. Using a fixed-size structure instead of a dynamically managed one (*e.g.,* a page table-like one) allows us to avoid the penalties involved with managing and accessing it. Note that while on 32-bit systems the size of the tagmap is reasonable, flat bitmaps are not practical on 64-bit architectures. For instance, in x86-64 a flat bitmap would require 32TB.

Byte-sized tags When libdft is using byte-sized tags, it stores them in dynamically allocated tagmap segments (see `tseg` in Figure 3). Every time the application gets a new chunk of memory implicitly by performing an image load (*e.g.,* when loading a dynamic shared object), or explicitly by invoking a system call like `mmap`, `mremap`, `brk`, and `shmat`, libdft intercepts the event and allocates an equally sized contiguous memory region. For instance, if the application requests an anonymous mapping of 1MB using `mmap`, libdft will "shadow" the allocated region with a tagmap segment of 1MB, for storing the byte tags of the `mmap`-ed memory. More importantly, tagmap segments that correspond to shared memory chunks are also shared. Hence, two processes running under libdft can effectively share shadow memory. To the best of our knowledge, we are the first to implement such a tag sharing scheme.

We obtain information about memory areas mapped at load time, or before libdft was attached on the application, through the `proc` pseudo-filesystem (`/proc/<pid>/maps`). This way we acquire the location of the stack and other kernel-mapped memory objects, such as the vDSO and vsyscall pages, and allocate the respective tagmap segments accordingly. In order to deal with the implicit expansion of the stack, libdft pre-allocates a tagmap segment to cover the stack as if it expands to its maximum value, which can be obtained via `getrlimit(RLIMIT_STACK)`. However, the same is not necessary for thread stacks, since they are allocated explicitly using `mmap`.

During initialization, libdft allocates a segment translation table (`STAB`) for mapping virtual addresses to their corresponding bytes in tagmap segments. Since memory is given to processes in blocks that are multiples of page size, `STAB` entries correspond to page size areas. For each page, `STAB` stores an addend value, which is effectively a 32-bit offset that needs to be added to all memory addresses inside that page, for retrieving the respective tag metadata. Assuming again a 3G/1G memory split and 4KB pages, `STAB` requires 3MB (*i.e.,* one entry for each 4KB in the range 0x00000000 – 0xBFFFFFFF). Whenever we allocate or free a tagmap segment, we update the `STAB` structure accordingly. Moreover, we ensure that segments that match with adjacent memory pages are also adjacent. This not only allows dealing with memory accesses crossing segment boundaries (*e.g.,* unaligned multi-byte accesses that span two pages are valid in x86), but also enables us to use a simple operation to retrieve the tag for any memory address: `taddr = vaddr + STAB[vaddr >> 12]`. The 20 MSBs of $vaddr$ are used as index in `STAB` to get an addend value, which in turn added to $vaddr$ itself for obtaining the respective tag bytes.

Byte-sized tags let us tag data using 8 different colors, but incur a higher per-byte memory overhead. Additionally, dynamically managing the respective tagmap segments also introduces overhead. However, we proactively allocate tagmap segments whenever the application maps new memory, instead of lazily waiting until a tagmap segment is used, to avoid the penalties involved with using branching instructions in analysis routines.

4.2 Code Instrumentation and Analysis

The *tracker* is the core component of libdft that is in charge of instrumenting a program to retrofit the DFT logic. It consists of two parts, shown in Figure 3.

4.2.1 Instrumentation Engine

The instrumentation engine is responsible for inspecting program instructions to determine the analysis routine(s) that should be injected for each case. We use Pin's instrumentation API to inspect every instruction before it is translated by the JIT compiler. We first resolve the instruction type (*e.g.,* arithmetic, move, logic), and then we analyze its operands for determining their category (*i.e.,* register, memory address, or immediate) and length (byte, word, double word). After gathering this information, we use Pin to insert the appropriate analysis routine before each instruction.

The instrumentation code is invoked once for every sequence of instructions, and the result (*i.e.,* the original code and analysis routines) is placed into Pin's code cache. We exploit code caching, by handling the x86 ISA complexity during the instrumentation phase, and keeping the analysis routines compact and fast. Specifically, we move the elaborate logic of discovering data dependencies and handling each variant of the same dependency category into the instrumentation phase. This allows us to aggressively optimize the propagation code by injecting compact, category-specific, and fast code snippets before each instruction. Due to the complexity and inherent redundancy of the ISA, our instrumentation engine consists of ∼3000 C++ lines of code (LOC).

4.2.2 Analysis Routines

The analysis routines contain the code that actually implements the DFT logic for each instruction. They are injected by the instrumentation engine before every instruction to assert, clear, and propagate tags, and unlike instrumentation code they execute more frequently (*i.e.,* the analysis code injected for a specific instruction, executes every time the instruction executes).

Carefully implementing these analysis routines is paramount for achieving good performance. For instance, while Pin tries to inline analysis code into the application's code, the use of branch instructions will cause it to insert a function call to the respective routines instead (recall that function calls require extra processing cycles). The same also stands for overly large analysis routines. *Interestingly, we observed that the number of instructions, excluding all types of jumps, which Pin can inline is ∼20.*

For these reasons, we introduce two guidelines for the development of efficient tag propagation code: (i) tag propagation should be branch-less, and (ii) 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. Our analysis routines are made up of ∼2500 C LOC, and include only arithmetical, logical, and memory operations. Moreover, we force Pin to use the *fastcall* x86 calling convention, for making the DFT code faster and smaller (*i.e.,* the compiler will avoid emitting push, pop, or stack-based parameter loading instructions).

Tracking code can be classified in the following categories based on the corresponding instruction type (the numbers in parentheses indicate the total number of analysis routines we implemented for each class, which are necessary for capturing the semantics of different operand types and sizes):

- *ALU* (21): analysis routines for the most common x86 instructions that typically have 2 or 3 operands, such as `ADD`, `SUB`, `AND`, `XOR`, `DIV`, `IMUL`, and so forth. For such instructions, we take the *union* of the source and destination operand tags, and we store the result in the respective tags of the destination operand(s). Immediates are always considered untagged.
- *XFER* (20): this class includes data transfers from a register to another register (*r2r*), from a register to a memory location and vice-versa (*r2m* and *m2r*), as well as from one memory address to another (*m2m*). For this type of instructions the source operand tags are copied to the destination operand tags, and again, immediates are always considered untagged.
- *CLR* (6): certain fairly complex instructions always result in their operands being untagged. Examples of such instructions include `CPUID`, `SETxx`, *etc.* Similarly, x86 *idioms* used for "zeroing" a register (*e.g.,*`xor eax, eax` and `sub eax, eax`), also result in untagging their operands.
- *SPECIAL* (45): this class includes analysis routines for x86 instructions that cannot be handled effectively with the aforementioned primitives, such as `XCHG`, `CMPXCHG`, `XADD`, `LEA`, `MOVSX`, `MOVZX`, and `CWDE`. For instance, although `XADD` can be handled by instrumenting the instruction twice with the respective XFER routines (for exchanging the tag values of the operands), and once with the ALU routine that handles `ADD`, the code size expansion would be prohibitive. Thus, we choose to implement an optimized analysis routine, for minimizing the injected code and inflicting less pressure on the code cache. libdft has one special handler for each quirky x86 instruction.
- *FPU, MMX, SSE*: these are ignored by default, unless their result is stored into one of the GPRs, or to a memory location. In such cases, the destination is untagged.

```
--------------------[r2r_alu_opb_l]--------------------
threads_ctx[tid].vcpu.gpr[dst] |=
    threads_ctx[tid].vcpu.gpr[src] & VCPU_MASK8;
--------------------[r2m_alu_opw]---------------------
*((uint16_t *)(mem_bitmap + VIRT2BYTE(dst))) |=
    (threads_ctx[tid].vcpu.gpr[src] & VCPU_MASK16) <<
        VIRT2BIT(dst);
--------------------[m2r_alu_opl]---------------------
threads_ctx[tid].vcpu.gpr[dst] |=
    (*((uint16_t *)(mem_bitmap + VIRT2BYTE(src))) >>
-------------------[r2r_xfer_opb_l]-------------------
threads_ctx[tid].vcpu.gpr[dst] =
    (threads_ctx[tid].vcpu.gpr[dst] & ~VCPU_MASK8) |
    (threads_ctx[tid].vcpu.gpr[src] & VCPU_MASK8);
--------------------[r2m_xfer_opw]--------------------
*((uint16_t *)(mem_bitmap + VIRT2BYTE(dst))) =
    (*((uint16_t *)(mem_bitmap + VIRT2BYTE(dst))) &
        ~(WORD_MASK << VIRT2BIT(dst))) |
    ((uint16_t)(threads_ctx[tid].vcpu.gpr[src] &
        VCPU_MASK16) << VIRT2BIT(dst));
--------------------[m2r_xfer_opl]--------------------
threads_ctx[tid].vcpu.gpr[dst] =
    (*((uint16_t *)(mem_bitmap + VIRT2BYTE(src))) >>
        VIRT2BIT(src)) & VCPU_MASK32;
```

Figure 4. Tag propagation code for various analysis routines when libdft is using bit-sized tags. The `VIRT2BYTE` macro is used for getting the byte offset of a specific address in `mem_bitmap` (it performs a bitwise right shift by 3), whereas `VIRT2BIT` gives the bit offset within the previously acquired byte.

```
--------------------[r2r_alu_opb_l]--------------------
*((uint8_t *)&threads_ctx[tid].vcpu.gpr[dst]) |=
    *((uint8_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[r2m_alu_opw]---------------------
*((uint16_t *)(dst + STAB[VIRT2STAB(dst)])) |=
    *((uint16_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[m2r_alu_opl]---------------------
threads_ctx[tid].vcpu.gpr[dst] |=
    *((uint32_t *)(src + STAB[VIRT2STAB(src)]));


-------------------[r2r_xfer_opb_l]-------------------
*((uint8_t *)&threads_ctx[tid].vcpu.gpr[dst]) =
    *((uint8_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[r2m_xfer_opw]--------------------
*((uint16_t *)&threads_ctx[tid].vcpu.gpr[dst]) =
    *((uint16_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[m2r_xfer_opl]--------------------
threads_ctx[tid].vcpu.gpr[dst] =
    *((uint32_t *)(src + STAB[VIRT2STAB(src)]));
```

Figure 5. Code snippets for different analysis routines when libdft is using byte-sized tags. `VIRT2STAB` is a macro for obtaining the `STAB` index given a virtual address (it performs a bitwise right shift by 12).

Figures 4 and 5 show excerpts from different types of analysis routines in the case of bit- and byte-sized tags, respectively. Code snippets labeled as `alu` correspond to routines that instrument 2 operand instructions belonging to the ALU category. On the contrary, `xfer` indicates propagation code for instructions of the XFER category. The operand size (*i.e.,* 8-, 16-, 32-bit) is designated by the `op{b, w, l}` label suffix, while the `r2r`, `r2m`, and `m2r` prefix is used for specifying the operand type (register vs. memory).

In the case of byte-sized tags, achieving single assignment tagmap updates and branch-less tag propagation is relatively easy, due to the design of our shadow memory. Specifically, if both operands are registers, then we merely need to perform a copy or a bitwise OR operation, of the appropriate size, between the respective GPRs in the `vcpu` structure of the current thread. On the other hand, if one of the operands is a memory location, the effective address (*i.e.,* `src` or `dst` depending on the instrumented instruction) goes through `STAB` for getting the addend value that should be added to the address itself, in order to address the tag bytes from the corresponding tagmap segment (see Section 4.1.2). The final propagation is performed similarly to the previous case.

Note that when libdft is configured to use bit-sized tags, analysis routines tend to be larger and more elaborate. This is due to the pedantic bit operations that are required, since we cannot simply "move" separate bits between different tagmap locations. Hence, to avoid using branch instructions or multiple assignment statements, we resort in bit masks and bitwise operations.

4.3 I/O Interface

The *I/O Interface* is the component of libdft that handles the exchange of data between the kernel and the process through system calls. In particular, it consists of two small pieces of instrumentation code, namely the `pre_syscall` and `post_syscall` stubs, and a table of system call meta-information (`syscall_desc[]`), as illustrated in Figure 3. The `syscall_desc` table holds specific libdft-related information for all the 344 system calls of the Linux kernel (up to v2.6.39). For instance, it stores user-registered callbacks (for using a system call as a data source or sink), descriptors for the arguments of the call, or whether the system call writes data to user space memory.

When a system call is made by the application, the stubs are called upon entry and exit. If the user has registered a callback function for a specific system call (either for entering or exiting), it is invoked. Otherwise, the default behavior of the `post_syscall` stub is to untag the data being written/returned by the system call. The advantages of this approach are twofold. First, we enable the tool writer to hook specific I/O channels (*e.g.,* network I/O streams) and perform selective tagging. This way, the developer can customize libdft by using system calls as data sources and sinks. Second, we eliminate tag leaks by taking into consideration that some system calls write specific data to user-provided buffers. For example, consider `gettimeofday` that upon each call overwrites user space memory that corresponds to one, or two, `struct timeval` data structures. Such system calls always result in sanitizing (*i.e.,* untagging) the data being returned, unless the tool writer has installed a callback that selectively tags the returned data. Finally, hooking a function call is straightforward, and can be performed directly using libdft's and Pin's API.

4.4 Optimizations

4.4.1 Fast `vcpu` Access (fast_vcpu)

libdft initially stored the per-thread `vcpu` structure in a global array indexed by Pin's virtual thread id (see Section 4.1.1). However, we determined that this was not the most efficient structure for this purpose. Because the array can be concurrently accessed from multiple threads, proper locking is required to safely expand it when new threads are created. Moreover, retrieving the `vcpu` from an analysis routine demands extra instructions (*i.e.,* array lookup).

Instead, we can utilize Pin's scratch registers to store a pointer to the `vcpu` structure of each thread. Scratch registers are thread-specific, virtual registers used internally by Pin, but also available for use in Pintools. Each time a new thread is created, we allocate a new `vcpu` structure as before, but instead of adding it in an array, we save its address in such a register. Analysis routines are modified to receive the scratch register pointing to the `vcpu` as argument, with `threads_ctx[tid].vcpu` changed to `thread_ctx->vcpu` (see Figures 4 and 5). This approach demands more register spilling, but as we experimentally confirmed (see Section 6), the benefits from avoiding locking and an extra array indexing operation outweigh the spilling overhead.

4.4.2 Fast REP-prefixed Instructions (fast_rep)

Certain x86 instructions, such as MOVS, STOS, and LODS, can be executed repeatedly using the REP prefix. During instrumentation, Pin treats them as implicit loops containing the un-prefixed instruction to allow Pintools insert analysis routines that receive the correct effective address (EA) used on each repetition. This introduces overhead because the REP-prefixed instruction is transformed to a loop, and because tag propagating code is executed within the loop.

However, the effective address (EA) used on each repetition depends on the EA used on the previous repetition and the value of the DF bit in the EFLAGS register: $EA = EA_{prev} \pm \{1, 2, 4\}$. We exploit this observation to move part of the analysis code outside the loop. Particularly, we perform the expensive mapping of a memory address to its shadow memory address only on the first repetition. The analysis routine handling the first loop, in addition to performing the required address mapping and propagating tags, caches the translated shadow memory address. Afterward, the analysis routine handling the rest of the repetitions can use the cached address to perform tag propagation faster. Note that moving all the propagation logic before the loop frequently has adverse effects, such as causing Pin to use a function call instead of inlining.

4.4.3 Huge TLB (huge_tlb)

DFT logic constantly operates on data tags. Both when using bit- and byte-sized tags, the analysis routines continuously access the mem_bitmap and STAB structures respectively. As the tag propagation code is interleaved with application instructions, memory accesses are spread between application and shadow memory, thus leading to poor performance of the CPU's translation lookaside buffer (TLB). We attempt to alleviate the problem by utilizing the multiple page size feature on x86 architectures for reducing the misses in the TLB. Specifically, by allocating mem_bitmap and STAB using the MAP_HUGETLB option with mmap (only for Linux kernel versions $\geq$ 2.6.32), the allocation is performed using pages of 4MB in size[1], effectively reducing TLB "poisoning" due to accesses in libdft's tagmap (*i.e.,* fewer TLB misses due to memory accesses in mem_bitmap and STAB, as well as fewer evictions of process entries for serving tagmap page faults).

4.4.4 Tagmap Collapse (tmap_col)

When using byte-sized tags, the tagmap allocates a page of memory for every page used by the application, including pages used for the heap, the stack, libraries, and so forth. However, not all pages are assigned the same access rights. For instance, code segments are usually write-protected, and the same is also true for certain other types of data (*e.g.,* constants, immutable objects, special-purpose shared memory segments). Therefore, unless explicitly altered by a tool writer, the tags corresponding to such pages will always be constant. We observe that the DFT logic will never need to legitimately alter the tag(s) for such memory pages, since the code of the application cannot legitimately update them either.

We exploit this observation, and significantly reduce the memory overhead of tagmap, by collapsing segments that correspond to write-protected memory regions into a single, constant segment. In particular, we allocate a special page using mmap, namely a *zero* page for clean tags, and set the access bits of that page to PROT_READ, effectively disallowing all writes to it. When new write-protected pages are mapped/allocated by the application, we update the corresponding STAB entries with addend values that map to the zero page. Note that for this optimization we also need to explicitly handle the mprotect system call for dealing with write-protected pages that are later, or temporarily, mapped as writeable, and vice-versa.

[1] If Physical Address Extension (PAE) is enabled, the size of large pages is set to 2MB.

4.5 Memory Protection

Dynamic DFT is frequently used to analyze malware or enforce security, and in that context it is desirable to guarantee the integrity of libdft by protecting its memory similarly to a sandbox [12]. As shown in Figure 2, the same address space is used by the application, Pin that allocates memory to store its code cache, and libdft that allocates the tagmap. Since all of the above reside inside the same space, the tracked program could accidentally or intentionally corrupt Pin or libdft.

Our solution to the problem is inspired by the scheme proposed by Xu *et al.* [28], and relies on the premise that application code cannot execute natively without first being instrumented and analyzed by libdft. Since we instrument all memory accesses, an instruction that tries to write at memory address $vaddr$ will be instrumented with the corresponding DFT logic, in order to assert or clear the respective tag(s) in tagmap. Therefore, by restricting access to specific blocks on the tagmap, we can prevent application code from accessing certain memory regions. In the case of bit-sized tags, we first enforce Pin to only use memory in a specific memory range (*e.g.,* in the lower 512MB, 0x00000000 – 0x20000000). Then, we proceed to protect the bits of mem_bitmap that correspond to that range (*e.g.,* the first 64MB). Whenever application code tries to access the lower 512MB of its virtual address space, the DFT analysis routines will access the protected blocks of the tagmap, leading to a memory violation error. However, note that the protected memory region cannot be arbitrary, and depends on the architecture's page size. Assuming a 4KB page size and 1-bit tags, both the beginning and ending of the protected memory range need to be aligned to 32K (8 bits per byte $\times$ 4KB page).

In the case of byte-sized tags, we allocate a *guard* page using mmap and set its access bits to PROT_NONE, effectively disallowing any access to it. During initialization, we set all STAB entries that correspond to unallocated memory pages to point to the guard page. From the application's perspective, Pin's and libdft's memory is always considered unallocated space. Hence, any access to these addresses will result in DFT logic operating on the guard page, leading again to a memory violation error. Note that by using libdft's API tool writers can register callback handlers for dealing with such violation errors.

5. Creating libdft-powered Tools

One of the most frequent incarnations of DFT has been dynamic taint analysis. DTA operates by tagging all data coming from the network as tainted, tracking their propagation, and alerting the user when they are used in a way that could compromise system 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 [3]), 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,INET6} socket descriptors. It also hooks

Function	Description
`libdft_init()`	Initialize the tagging engine
`libdft_start()`	Commence execution
`libdft_die()`	Detach from the application
`ins_set_pre()` `ins_set_post()` `ins_set_clr()`	Register instruction callbacks to be invoked before, after, or instead libdft's instrumentation
`syscall_set_pre()` `syscall_set_post()`	Hook a system call entry or return
`tagmap_set{b,w,l}()` `tagmap_setn()`	Tag {1, 2, 4} and `n` bytes of virtual memory

Table 1. Overview of libdft's API.

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 receive 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 `execve` system call for tainted parameters.

Besides DTA, we have used libdft in other settings. Specifically, we used libdft for studying a set of optimization techniques that minimize the runtime slowdown of dynamic DFT, by combining dynamic and static analysis for separating program logic from tracking logic and eliminate needless tracking [14]. Our results not only indicate that libdft can be used as the apparatus for studying novel optimization methodologies, but also demonstrate the applicability of our framework to a different domain. Finally, colleagues that were given access to early versions of libdft used it for building Taint-Exchange [29], a generic cross-process and cross-host taint tracking tool that allows studying the interaction of distributed components, by multiplexing fine-grained taint information into I/O channels. Additional applications of libdft are in progress, which will help us validate and refine our API as needed.

6. Evaluation

We evaluated libdft using a variety of software including a web and database (DB) server, command-line and network utilities, a web browser, and the SPEC CPU2000 suite. Our aim is to quantify the performance of libdft, and establish a set of bounds for the various types of applications and software complexity. To the best of our knowledge, we are the first to evaluate the performance of a dynamic DFT framework in such depth. We also compare the performance of libdft with the results reported in selected related studies, and proceed to evaluate the effectiveness of the various optimizations and design decisions we took. We close this section with an evaluation of the performance and effectiveness of the libdft-powered DTA tool presented earlier.

The results presented throughout this section are mean values, calculated after running 10 repetitions of each experiment, while the reported confidence intervals correspond to 95%. Our testbed consisted of two identical hosts, equipped with two 2.66 GHz quad core Intel Xeon X5500 CPUs and 24GB of RAM each, running Linux (Debian "squeeze" with kernel version 2.6.32). The version of Pin used during the evaluation was 2.9 (build 39586). While conducting our experiments all hosts were idle.

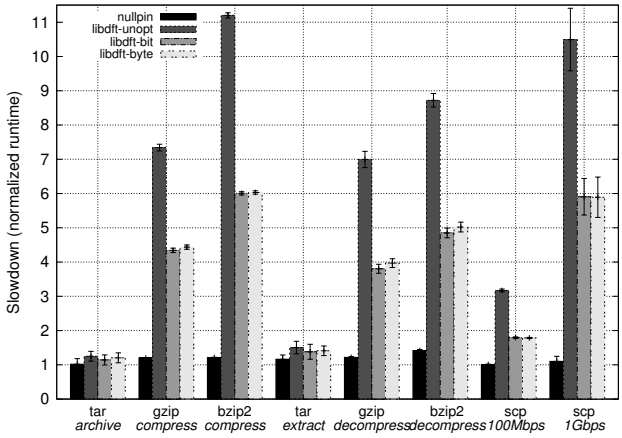


Figure 6. The slowdown imposed by Pin and libdft when running four common Unix command-line utilities.

6.1 Performance

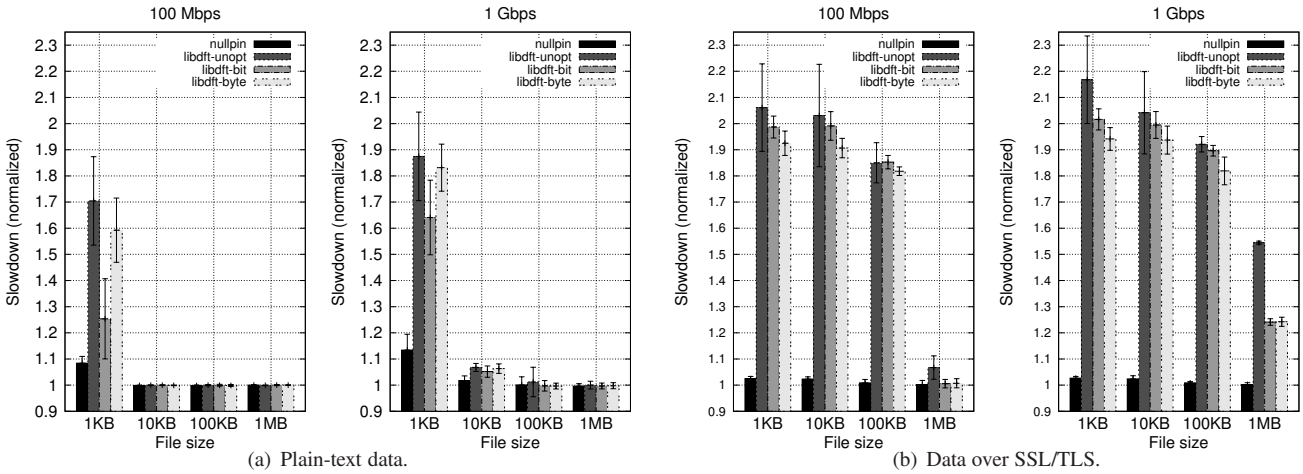
We developed four simple Pintools to aid us in evaluating libdft. The first is *nullpin*, which essentially runs a process using Pin without any form of instrumentation or analysis. This tool measures the overhead imposed by Pin's runtime environment alone. The second and third, namely *libdft-*{*bit, byte*}, utilize libdft for applying DFT on the application being executed, and use bit-sized and byte-sized tags respectively. These tools measure the overhead of libdft when employing single assignment and branch-less propagation, plus the optimizations presented in Section 4.4. Finally, in order to demonstrate the efficacy of our design choices, we also evaluated *libdft-unopt*, which approximates the behavior of an unoptimized DFT framework. Specifically, it does not employ any optimization scheme and its analysis routines are not inlined, effectively resembling the impact of lax tag propagation.[2] Note that since the performance of libdft does not depend on the existence or amount of tagged data, none of our tools uses any of the API functions for customizing the applied DFT.

Utilities The goal of our first benchmark was to quantify the performance of libdft with commonly used Unix utilities. For this experiment, we used the GNU versions of `tar`, `gzip`, and `bzip2`, as well as `scp` from the OpenSSH package. We selected these applications because they represent different workloads. `tar` performs mostly I/O, while `gzip` and `bzip2` are CPU-bound applications. In between, `scp` is both I/O driven and CPU intensive.

We run all the tools natively, and using our four tools, and measured their execution time with Unix `time` utility. We used `tar` for archiving and extracting a vanilla Linux kernel "tarball" (v2.6.35.6; ∼400MB), whereas `gzip` and `bzip2` were used for compressing and decompressing it respectively. For `scp`, we copied 1GB of randomly generated data over SSH, first over an 100Mbps link and then over an 1Gbps link. We present the results in Figure 6.

libdft-bit imposes a slowdown that ranges between 1.14x and 6x (average 3.65x), while libdft-byte ranges between 1.20x and 6.03x (average 3.72x). Pin alone imposes an 1.17x slowdown (this is Pin's baseline). Overall, the more CPU-bound an application is, the larger the impact of libdft. For instance, `bzip2` is the most CPU intensive and `tar` the least, representing the worst and best performance of libdft. I/O operations have a positive effect on DFT. This is also confirmed by the overhead of `scp` over an 1Gbps link, which is higher when compared with the 100Mbps case.

[2] The body of the analysis routines used in libdft-unopt remains highly condensed, since they are the same routines that we use in libdft-{bit, byte}. Hence, the overhead measured with this tool gives a lower bound of an unoptimized DFT implementation.



(a) Plain-text data. (b) Data over SSL/TLS.

Figure 7. The slowdown incurred by Pin and libdft on the Apache web server when serving static HTML files of different sizes. We used Apache's benchmarking utility *ApacheBench (ab)* to measure the mean time per request for all four files, first over an 100Mbps link, and then over an 1Gbps link.

We surmise that nullpin and libdft perform worse when the limiting factor is not the I/O, because the respective latency hides the translation and tag propagation overhead. Indeed, when utilizing the 1Gbps link, the bottleneck shifts to the CPU, greatly reducing `scp`'s throughput. Also note that byte-sized tags impose an additional overhead of 1.9%, which stems from the management cost of the tagmap segments and memory shadowing (see Section 4.1.2). Compared with an unoptimized implementation, libdft with all our optimizations performs 3.64% to 46.37% faster.

Apache The second set of experiments calculates the performance slowdown of libdft when applied on larger and more complex software. Specifically, we investigate how libdft behaves when instrumenting the commonly used Apache web server. We used Apache v2.2.16 and configured it to pre-fork all the worker processes (pre-forking is a standard multiprocessing Apache module), in order to avoid high fluctuations in performance, due to Apache forking extra processes for handling the incoming requests at the beginning of our experiments. All other options were left to their default setting. We measured Apache's performance using its own utility `ab` and static HTML files of different size. In particular, we chose files with sizes of 1KB, 10KB, 100KB, and 1MB, and once again run the server natively and with our four tools. We also tested Apache over different network links, as well as with and without SSL/TLS encryption.

Figure 7(a) illustrates the results for running Apache, and transferring data in plaintext. We observe that as the size of the file being served increases, libdft's overhead diminishes. Similarly to our previous experiment, the time Apache spends performing I/O hides our overhead. As a result, libdft has negligible performance impact when Apache is serving files larger than 10KB at 100Mbps and 100KB at 1Gbps. In antithesis, libdft imposes an 1.25x/1.64x slowdown with 1KB files at 100Mbps/1Gbps when using bit-sized tags. The overhead of byte-sized tagging becomes more evident with smaller files because more requests are served by Apache, also increasing the number of `mmap` calls performed. This leads to higher tagmap management overhead, as segments need to be frequently allocated and freed. We anticipate that we can amortize this extra overhead by releasing segments more lazily, as we may have to re-allocate them soon after.

Figure 7(b) shows the results of conducting the same experiments, but this time using SSL/TLS encryption. We notice that libdft has larger impact when running on top of SSL. In the 100Mbps scenario, the slowdown becomes negligible only for files larger than 1MB, whereas at 1Gbps libdft imposes an 1.24x slowdown even with 1MB files. The reason behind this behavior is that the intensive cryptographic operations performed by SSL make the server CPU-bound.

Interestingly, libdft-byte performs 3% better (on average) than libdft-bit. In order to better understand this behavior, we analyzed the mix of the executed instructions and observed that serving a web page over SSL results in an increased number of XFER-type instrumentations, where one of the two operands is memory.[3] *Tag propagation code that corresponds to data transfers is more expensive in the case of bit-sized tags than the case of byte-sized tags.* This is because the body of the respective *r2m*, *m2r*, and *m2m* analysis routines contains more instructions, due to the elaborate bit operations that are necessary for asserting only specific bits in the tagmap (see Figure 4 and 5 in Section 4.2.2).

MySQL In Figure 8(a), we present the results from evaluating MySQL DB server. We used MySQL v5.1.49 and its own benchmark suite (`sql-bench`). The suite consists of four different tests, which assess the completion time of various DB operations like table creation and modification, data selection and insertion, *etc.* We notice that the average slowdown incurred by Pin's instrumentation alone is 1.64x, which is higher than the overhead observed when running smaller utilities (see Figure 6). The increased overhead is attributed to the significantly larger size of MySQL's codebase, which applies pressure on Pin's JIT compiler and code cache.

As far as libdft is concerned, the average slowdown on the five test suites was 3.36x when using byte-sized tags, and 3.55x when using bit-sized tags. Similarly to our previous experiments, libdft's overhead became more pronounced with more complex and CPU intensive tasks. In this case, the `test-insert` benchmark was the most exhaustive, involving table creations, random-ordered row insertions, duplicates checking, ordered selection, deletion, and so forth, and exhibited the largest slowdown (4.65x/4.83x). Note that libdft performs 5.65% faster when using byte-sized tags. Again, we analyzed the instructions of MySQL and observed a significant amount of XFER-type tag propagation routines in the mix. Our combined set of optimizations reduces the runtime overhead by 20.58% – 24.83%.

[3] SSL makes heavy use of instructions like `MOVS`, `STOS`, `MOVSX`, and `MOVZX`.

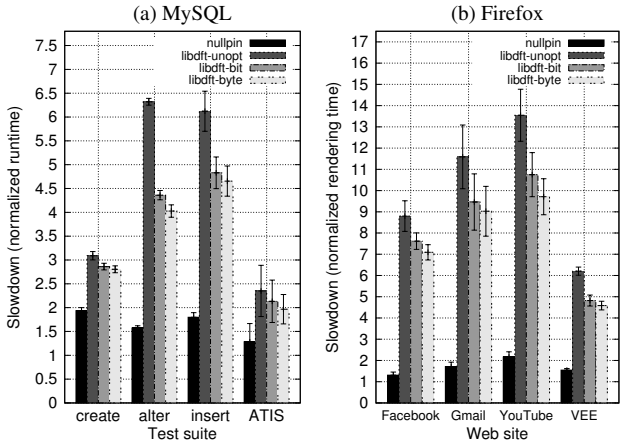


Figure 8. The overhead of Pin and libdft when running MySQL and Firefox. We employed MySQL's *sql-bench* benchmark suite, which measures table creation, data selection and insertion. Regarding Firefox, we accessed the 3 most popular web sites from Alexa's Top 500 and the VEE web site.

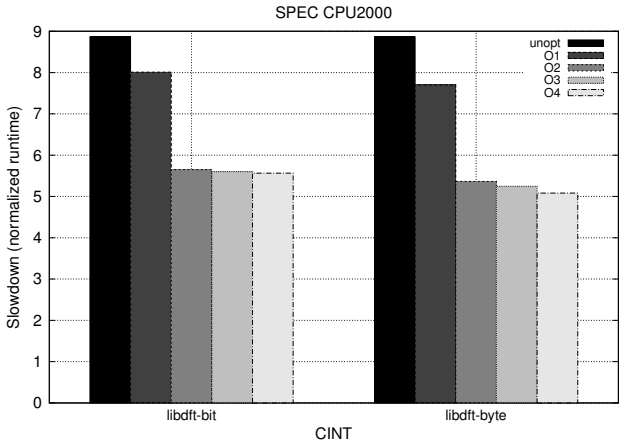


Figure 9. The impact of our optimizations when applied cumulatively on the SPEC CPU2000 benchmark: O1 (branch-less and single assignment tag propagation), O2 (O1 + `fast_vcpu`), O3 (O2 + `fast_rep`), O4 (O3 + `huge_tlb`).

Firefox After evaluating two of the most popular servers, we tested libdft with the Firefox web browser that has even larger and more complex codebase, and complements our evaluation of client-side software that started with the smaller utilities. We used Mozilla Firefox v3.6.18 to access the three most popular web sites according to Alexa's Top 500 (`http://www.alexa.com/topsites`), as well as the site of VEE, and measured the effect of libdft on rendering time. Figure 8(b) illustrates the results for this experiment. We see that the average slowdown under nullpin, libdft-bit, and libdft-byte, was 1.68x, 8.16x, and 7.06x respectively. The overhead is relatively low when accessing mostly static content web pages (VEE), while it increases significantly when accessing media-rich sites (YouTube), and sites that depend heavily on JavaScript (JS) like Gmail. Note that for Facebook and Gmail, we accessed and measured a real profile page and not the log in screen. The effect of optimizations ranges between 18.64% and 29.68%.

Next, we benchmarked the JS engine of Firefox using the Dromaeo test suite (`http://dromaeo.com/?recommended`). Dromaeo measures the speed of specific JS operations, so there is no I/O or network lag, as in the page loading benchmark. We observed that the slowdown incurred by both nullpin and libdft was considerably higher, with an average of 3.2x for nullpin, and 14.52x/13.9x for libdft-bit/libdft-byte respectively. While increased overhead was expected because this benchmark is also CPU-bound, the slowdown is significantly larger from previous CPU intensive experiments. *It seems that since JS is interpreted and subsequently "jitted" by the browser's runtime, it interferes with the translation and optimizations performed by Pin's JIT engine, thus leading to excessive runtime overheads.* In fact, this implies that significant overheads would emerge whenever DBI is combined with an interpreting runtime environment such as JS. Regardless, when using the browser to access common web sites, we do not suffer the performance decline observed in the Dromaeo benchmark.

6.2 Effectiveness of Optimizations

In order to quantify the impact of our set of optimizations presented in Section 4, we used the SPEC CPU2000 suite.

Figure 9 shows the overhead of running the suite under the unoptimized version of libdft (libdft-unopt) normalized to native execution, and the improvement in performance contributed by each optimization as it is incorporated in our framework. Branch-less and single assignment tag propagation, denoted by O1, has a notable impact on the imposed overhead, reducing it by 10.65% when libdft uses bit-sized tags and 15.09% in case of byte-sized tags. This optimization captures the impact of our efforts for aggressively inlining the analysis routines implementing tag propagation. Optimization O2 (O1 + `fast_vcpu`) reduces running time by 29.45% and 30.34% for libdft-bit and libdft-byte respectively. Recall that `fast_vcpu` results in smaller and lock-free code. O3 and O4, which add `fast_rep` and `huge_tlb` respectively, offer marginally enhanced performance to libdft (*i.e.,* 0.8% and 0.6% with libdft-bit, and 2.17% and 3.15% with libdft-byte). This is because SPEC CPU2000 does not utilize large memory chunks, nor it contains a significant amount of REP-prefixed instructions.

Application	Vulnerability	SecurityFocus ID
wu-ftpd v2.6.0	Format string	BID 1387
ATPhttpd v0.4	Stack overflow	BID 5215
WsMp3 v0.0.2	Heap corruption	BID 6240
ProFTPD v1.3.3	Stack overflow	BID 44562

Table 2. libdft-DTA successfully prevented the listed attacks.

6.3 The libdft-DTA Tool

The purpose of developing the libdft-DTA tool was not to provide a solid DTA solution, but to demonstrate that our API can be used to easily and quickly develop otherwise complex tools. Nevertheless, we tested its effectiveness using the set of exploits listed in Table 2. In all cases, it successfully detected and prevented the exploitation of the corresponding application. We also evaluated the performance of the tool using `scp`, Apache, and MySQL, and compare it with libdft's baseline performance. We observe that the additional overhead imposed by the tool is negligible ($\leq$ +7%) over the baseline overhead of DFT. While we cannot claim that all libdft-based tools will have such low additional overhead, our results demonstrate that libdft can be customized to implement problem-specific instances of DFT efficiently.

7. Limitations and Future Work

Thread safety As we discussed in Section 4.1.2, when libdft is configured to use bit-sized tags, it coalesces the tags of 8 consecutive byte memory locations into a single byte in `mem_bitmap`.

However, since there is no synchronization among threads that update `mem_bitmap` in parallel, there may be a race condition. More specifically, in order to handle unaligned multi-byte memory accesses that are valid in x86, the analysis routines of libdft-bit implement the single assignment tag propagation scheme using 16-bit words (see Figure 4). Therefore, if more than one threads are concurrently updating memory locations that are less than 16 bytes apart, the respective analysis routines of libdft will operate on the same 16-bit word on tagmap, leading into clobbered results.

Although such cases are possible, our assumption is that they are very rare. An intuition that is also supported by the fact that compilers align most variables to 4 or even 8 bytes. During our experimental evaluation, we verified (by tracing memory accesses) that multithreaded applications like Firefox, MySQL, and Apache, do not access memory in such a way. Nevertheless, this was one of the reasons for implementing a shadowing scheme that supports larger tag sizes (byte-sized), which do not suffer from such issues.

Device control In Linux, device control is mostly performed using the system call `ioctl`. This call uses a command number that identifies the operation to be performed, and in certain cases the receiver in the kernel. While attempts have been made to apply a formula on this number, due to backward compatibility issues and programmer errors, actual ioctl numbers are many times arbitrary (interested readers are referred to `ioctl_list(2)`). Identifying individual `ioctl` calls is necessary to sanitize the memory locations where data are being read. While `ioctl` is not used by regular applications, later versions of libdft could utilize system call tracing tools, such as `strace`, for pinpointing ioctls.

Future considerations In the future, we plan to extend libdft to run on Windows OSs and 64-bit architectures. We expect that porting libdft to run on Windows will be straightforward. Currently, all of libdft's components, besides the system call part of the I/O interface, can be used "as is" with the Windows OS (the underlying ISA is the same). Note that although the Windows system calls are numerous, they are rarely used directly by developers. libdft needs to intercept them solely for sanitizing the data being read into the process. Also, porting our framework to work with 64-bit x86 architectures does not pose any significant challenges, but requires moderate engineering effort. Finally, we plan to further investigate the impact of the `tmap_col` and `huge_tlb` optimizations. Specifically, we seek to quantify the reduction in memory consumption of specific types of applications due to tagmap collapse, as well as identify cases where fewer TLB misses can speedup libdft.

8. Related Work

Dytan [6], Minemu [2], LIFT [22], and PTT [11], are previously proposed dynamic DFT systems that are highly related to libdft. Dytan is the most flexible tool, allowing users to customize its sources, sinks, and propagation policy, while it can also track data based on control-flow dependencies (see Section 2; Figure 1). Albeit flexible, it incurs high performance penalties. For instance, `gzip` executes 30x slower than running natively, even when implicit tracking is turned off. When the latter is utilized, `gzip` performs 50x slower, while it can also lead to taint explosion. That is, erroneously tracking large amounts of data, due to the imprecision of control-flow data dependencies [23].

In contrast, Minemu is the fastest tool, but it provides limited functionality. It uses an ad hoc emulator for the mere purpose of performing fast DTA, and cannot be configured for use in other domains without modifying the emulator itself. Moreover, it does not provide colored tags, nor it supports self-modifying code. More importantly, Minemu cannot be used "as-is" on 64-bit architectures, due to its shadow memory design and heavy reliance on SSE (XMM) registers. In particular, it exploits the XMM registers to avoid spilling GPRs during taint tracking. However, this optimization may not have the same benefits on x86-64, since SSE has become standard and Minemu will have to resort on XMM spilling.

LIFT is another low-overhead dynamic DFT system. Unlike libdft, it focuses on detecting attacks instead of providing an extensible framework. LIFT builds on the premise that programs frequently execute without tagged data, and as a result tag propagation can be omitted. Their approach involves activating data tracking, only when tagged data are loaded in one of the CPU registers. This optimization is orthogonal to libdft, which could also benefit from dynamically enabling and disabling data tracking. In practice, the speedup depends on the program at hand and how often it operates on tagged data. Note that with tracking always enabled, LIFT exhibits higher overhead than libdft when running the SPECint benchmark. Operation-wise, it only runs with x86-64 binaries, and it does not support multithreading by design.

PTT is different from the rest of the systems because it performs system-wide data tracking, and uses additional cores through parallelization to improve performance. Even though it is one of the fastest such systems, it still incurs significant overheads.

TaintCheck [18] was one of the first tools to utilize DTA, for protecting binary-only software from buffer overflow and other types of memory corruption attacks, entirely in software. TaintCheck incurs prohibitive slowdowns, which may cause an application to run up to 37x slower. Vigilante [7] utilizes DFT to generate self-certifying alerts (SCAs). These alerts accurately describe the network message that was used in a previously detected attack, and can be distributed to parties hosting vulnerable software without the need for a secure distribution channel, because they can be independently and securely certified by the receiving party. Vigilante relies on the possibility of capturing attacks on server honeypots that will generate the SCAs, and as such does not focus on performance.

Eudaemon [20] builds upon the Qemu user space emulator to allow one selectively apply taint analysis on a process (*e.g.,* when potentially harmful actions are taken, or the system is idle). It incurs an overhead of approximately 9x, which can be alleviated on long-running applications by selectively enabling or disabling DFT. TaintEraser [31] uses Pin to apply taint analysis on binaries for the purpose of identifying information leaks. Furthermore, it introduces the concept of "function shortcuts" to reduce the overhead. These shortcuts enable the native execution of frequently called functions, but need to be defined manually by the developer, reducing the portability and practicality of the system. libdft could also benefit by such shortcuts (*e.g.,* in prevalent libc functions).

Hardware implementations of DFT [8, 9, 24] have been proposed to evade the large penalties imposed by software-only implementations, but unfortunately they have had no appeal with hardware vendors. Implementations of DTA using virtual machines and emulators have been also proposed [5, 13, 21]. While these solutions have some practicality for malware analysis platforms and honeypots, they induce slowdowns that make them impractical on production systems.

Speck [19] increases the performance of security checks, like DTA, by decoupling the checks from application execution using process-level replaying and Pin. In this manner, DTA can be run parallel with the application, while the DTA itself can be also parallelized. Similarly, Aftersight [4] decouples VM execution from program analysis, offloading the task to a separate, or even remote, platform. HiStar [30] uses labels to tag and protect sensitive data. It is a new OS design, and its main focus is to protect the OS from components that start exhibiting malicious behavior after being compromised. HiStar's data tracking is more coarse-grained than libdft's, and introduces major changes in the OS level.

9. Conclusions

We presented libdft, a practical dynamic DFT platform. Our goal is to facilitate future research by providing a framework that is at once *fast*, *reusable*, and applicable to *commodity software and hardware*. We also investigated the reasons that DFT tools based on DBI frameworks frequently perform badly, and presented the practices that need to be avoided by the authors of such tools. Our evaluation shows that libdft imposes low overhead, comparable to or faster than previous work. Its effect on web server throughput can be negligible when running over a 100Mbps network link and serving static HTML files, while even when switching to a 1Gbps link it never exceeds 2.04x. Moreover, we showed that performance depends on application CPU and I/O requirements. For instance, MySQL performs approximately 3.36x slower under libdft, while I/O intensive tools like `tar` sustain less than 1.14x overhead. We believe that libdft strikes a balance between usability and performance, incurring non prohibitive costs, even when running large and complex software like MySQL, Apache, and Firefox. Our implementation is freely available at: `http://www.cs.columbia.edu/~vpk/research/libdft/`

Acknowledgments

We thank Michalis Polychronakis, Dimitris Geneiatakis, and our shepherd, David Gregg, for providing comments on earlier drafts of this paper. This work was supported by DARPA, the US Air Force and the National Science Foundation through Contracts DARPA-FA8750-10-2-0253 and AFRL-FA8650-10-C-7024, and Grant CNS-09-14312, respectively, with additional support from Google and Intel Corp. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, the Air Force, NSF, Google or Intel.

References

[1] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. of the* 9^{th} *OSDI*, pages 237–250, 2010.

[2] E. Bosman, A. Slowinska, and H. Bos. Minemu: The World's Fastest Taint Tracker. In *Proc. of the* 14^{th} *RAID*, pages 1–20, 2011.

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proc. of the* 17^{th} *CCS*, pages 559–572, 2010.

[4] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of the 2008 USENIX ATC*, pages 1–14.

[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proc. of the* 13^{th} *USENIX Security*, pages 321–336, 2004.

[6] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proc. of the 2007 ISSTA*, pages 196–206.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proc. of the* 20^{th} *SOSP*, pages 133–147, 2005.

[8] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. of the* 37^{th} *MICRO*, pages 221–232, 2004.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Real-World Buffer Overflow Protection for Userspace & Kernelspace. In *Proc. of the* 17^{th} *USENIX Security*, pages 395–410, 2008.

[10] 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 *Proc. of the* 9^{th} *OSDI*, pages 393–407, 2010.

[11] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley. Towards Practical Taint Tracking. Technical Report UCB/EECS-2010-92, EECS Dept., University of California, Berkeley, USA, 2010.

[12] B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proc. of the 2008 USENIX ATC*, pages 293–306.

[13] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-based Protection using Demand Emulation. In *Proc. of the 2006 EuroSys*, pages 29–41.

[14] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *Proc. of the* 19^{th} *NDSS*, 2012.

[15] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proc. of the* 18^{th} *NDSS*, 2011.

[16] 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 *Proc. of the 2005 PLDI*, pages 190–200.

[17] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. of the* 26^{th} *POPL*, pages 228–241, 1999.

[18] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the* 12^{th} *NDSS*, 2005.

[19] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proc. of the* 13^{th} *ASPLOS*, pages 308–318, 2008.

[20] G. Portokalidis and H. Bos. Eudaemon: Involuntary and On-Demand Emulation Against Zero-Day Exploits. In *Proc. of the 2008 EuroSys*, pages 287–299.

[21] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of the 2006 EuroSys*, pages 15–27.

[22] 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 *Proc. of the* 39^{th} *MICRO*, pages 135–148, 2006.

[23] A. Slowinska and H. Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proc. of the 2009 EuroSys*, pages 61–74.

[24] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proc. of the* 11^{th} *ASPLOS*, pages 85–96, 2004.

[25] 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 *Proc. of the* 37^{th} *MICRO*, pages 243–254, 2004.

[26] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A Programmable Accelerator for Dynamic Taint Propagation. In *Proc. of the* 14^{th} *HPCA*, pages 173–184, 2008.

[27] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proc. of the* 31^{st} *IEEE S&P*, pages 497–512, 2010.

[28] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. of the* 15^{th} *USENIX Security*, pages 121–136, 2006.

[29] A. Zavou, G. Portokalidis, and A. D. Keromytis. Taint-Exchange: A Generic System for Cross-process and Cross-host Taint Tracking. In *Proc. of the* 6^{th} *IWSEC*, pages 113–128, 2011.

[30] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proc. of the* 7^{th} *OSDI*, pages 263–278, 2006.

[31] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, 2011.