libdft: Dynamic Data Flow Tracking for the Masses

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@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. `libdft` 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. `libdft` 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], retrofitted 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.

`libdft` argues that a *practical* DFT implementation should be concurrently (i) *fast*, (ii) *reusable*, and (iii) *applicable to commodity hardware and software*. `libdft` 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.

ACSAC '22, December 5–9, 2022, Austin, TX, USA
2022. ACM ISBN XXX-X-XXXX-XXXX-X/20/12…$6.66
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
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);
```

Figure 1: Example of code with data dependencies.

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, `libdft` 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

`libdft` 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 simple 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 presence 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/loading time), or *dynamically* using virtualization or dynamic binary instrumentation (DBI).

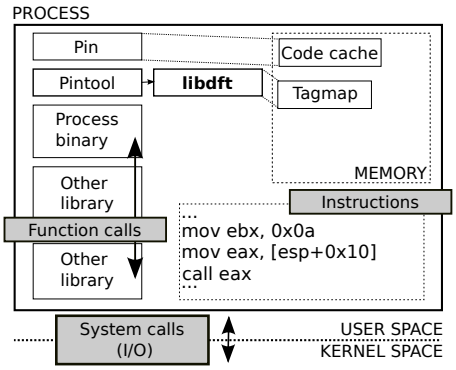

Figure 2: Process image of a binary running under `libdft`. The highlighted boxes describe possible data sources and sinks that can be used with `libdft`.

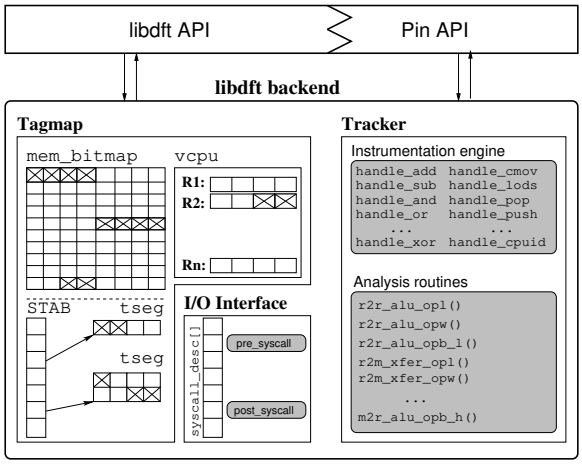

Figure 3: The architecture of `libdft`. The shaded components illustrate the instrumentation and analysis code that implements the DFT logic, whereas the x-marked regions on the tagmap indicate tagged bytes.

Static systems apply DFT by recompiling software using a modified compiler [48] or binary rewriter [8], or a source-to-source transformation engine [51]. Conversely, the dynamic ones can be directly applied on unmodified binaries, including commercial off-the-shelf software [27, 42, 54]. In both cases, software needs to be extensively instrumented for associating data with some kind of tag and 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 slower than static ones, have the advantage of being immediately, and incrementally, applicable to already deployed (or binary-only) software.

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.

```
--------------------[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_op1]---------------------
threads_ctx[tid].vcpu.gpr[dst] |=
    (*((uint16_t *)(mem_bitmap + VIRT2BYTE(src))) >>
        VIRT2BIT(src)) & VCPU_MASK32;
-------------------[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_op1]--------------------
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`, whereas `VIRT2BIT` gives the bit offset within the previously-acquired byte.

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 arithmetical, 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, fast_rep, 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.

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 the `libdft` API.

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,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 `execve` 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 (VUzzer [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://gitlab.com/brown-ssl/libdft/

REFERENCES

[1] Dennis Andriesse. 2018. *Practical Binary Analysis*. No Starch Press.
[2] AngoraFuzzer. [n. d.]. `libdft64`. https://github.com/AngoraFuzzer/libdft64.
[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDoid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.
[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 307–320.
[5] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. *ACM SIGPLAN Notices* 49, 10 (2014), 83–101.
[6] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*. 1–20.
[7] Alexandre Bouard, Maximilian Graf, and Dennis Burgkhardt. 2013. Middleware-based Security and Privacy for In-car Integration of Third-party Applications. In *IFIP International Conference on Trust Management (TM)*. 17–32.
[8] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SelectiveTaint: Efficient Data Flow Tracking with Static Binary Rewriting. In *USENIX Security Symposium (SEC)*. 1665–1682.

[9] Xi Chen, Asia Slowinska, and Herbert Bos. 2013. Who Allocated my Memory? Detecting Custom Memory Allocators in C Binaries. In *Working Conference on Reverse Engineering (WCRE)*. 22–31.

[10] Xi Chen, Asia Slowinska, and Herbert Bos. 2016. On the Detection of Custom Memory Allocators in C Binaries. *Empirical Software Engineering (ESE)* 21, 3 (2016), 753–777.

[11] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium (SEC)*. 321–336.

[12] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve'em All: Inferring Taint Rules without Architectural Semantics. In *Network and Distributed System Security Symposium (NDSS)*.

[13] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. 2006. Minos: Architectural Support for Protecting Control Data. *ACM Transactions on Architecture and Code Optimization (TACO)* 3, 4 (2006), 359–389.

[14] Baojiang Cui, Fuwei Wang, Tao Guo, Guowei Dong, and Bing Zhao. 2013. FlowWalker: A Fast and Precise Off-line Taint Analysis Framework. In *International Conference on Emerging Intelligent Data and Web Technologies (EIDWT)*. 583–588.

[15] Yu Ding. [n. d.]. `dftwin`. https://github.com/dingelish/dftwin.

[16] Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. 2020. On the Dissection of Evasive Malware. *IEEE Transactions on Information Forensics and Security (TIFS)* 15 (2020), 2750–2765.

[17] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[18] John Galea and Daniel Kroening. 2020. The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation. In *ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*. 622–636.

[19] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. 2012. Adaptive Defenses for Commodity Software through Virtual Application Partitioning. In *ACM Conference on Computer and Communications Security (CCS)*. 133–144.

[20] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. 2016. DECAF: A Platform-neutral Whole-system Dynamic Binary Analysis Platform. *IEEE Transactions on Software Engineering* 43, 2 (2016), 164–184.

[21] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. 2006. Practical Taint-based Protection using Demand Emulation. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 29–41.

[22] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using Input Type Inference to Improve Fuzzing. In *Annual Computer Security Applications Conference (ACSAC)*. 505–517.

[23] Kangkook Jee, Vasileios P. Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *ACM Conference on Computer and Communications Security (CCS)*. 235–246.

[24] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D. Keromytis. 2012. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *Network and Distributed System Security Symposium (NDSS)*.

[25] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation with On-demand Inter-process Information Flow Tracking. In *ACM Conference on Computer and Communications Security (CCS)*. 377–390.

[26] Vasileios P. Kemerlis. [n. d.]. `libdft`. https://gitlab.com/brown-ssl/libdft/.

[27] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*. 121–132.

[28] Benjamin Kollenda, Enes Göktaş, Tim Blazytko, Philipp Koppe, Robert Gawlik, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, and Thorsten Holz. 2017. Towards Automated Discovery of Crash-resistant Primitives in Binary Executables. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 189–200.

[29] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 503–515.

[30] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-language Dynamic Information Flow Analysis. In *USENIX Security Symposium (SEC)*. 2513–2530.

[31] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic Runtime Error Repair and Containment via Recovery Shepherding. *ACM SIGPLAN Notices* 49, 6 (2014), 227–238.

[32] Enrico Lovat, Alexander Fromm, Martin Mohr, and Alexander Pretschner. 2015. SHRIFT: System-wide Hybrid Information Flow Tracking. In *IFIP International Information Security and Privacy Conference (SEC)*. 371–385.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices* 40, 6 (2005), 190–200.

[34] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. Straighttaint: Decoupled Offline Symbolic Taint Analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 308–319.

[35] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *USENIX Security Symposium (SEC)*. 65–80.

[36] Andrew C. Myers. 1999. JFlow: Practical Mostly-static Information Flow Control. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. 228–241.

[37] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*.

[38] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *USENIX Security Symposium (SEC)*. 121–138.

[39] Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. 2013. CloudFence: Data Flow Tracking as a Cloud Service. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 411–431.

[40] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. 2006. Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 15–27.

[41] Weizhong Qiang, Yingda Huang, Deqing Zou, Hai Jin, Shizhen Wang, and Guozhong Sun. 2017. Fully Context-sensitive CFI for COTS binaries. In *Australasian Conference on Information Security and Privacy (ACISP)*. 435–442.

[42] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 135–148.

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*.

[44] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking Inside the Black-box: Capturing Data Provenance using Dynamic Instrumentation. In *International Provenance and Annotation Workshop (IPAW)*. 155–167.

[45] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. *ACM SIGPLAN Notices* 39, 11 (2004), 85–96.

[46] Zhenzhou Tian, Qinghua Zheng, Ting Liu, Ming Fan, Eryue Zhuang, and Zijiang Yang. 2015. Software Plagiarism Detection with Birthmarks based on Dynamic Key Instruction Sequences. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1217–1235.

[47] Daniel Townley, Khaled N Khasawneh, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Lei Yu. 2019. LATCH: A Locality-aware Taint CHecker. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 969–982.

[48] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A. Reis, Manish Vachharajani, and David I. August. 2004. RIFLE: An Architectural Framework for User-centric Information-flow Security. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 243–254.

[49] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *ACM Conference on Computer and Communications Security (CCS)*. 1675–1689.

[50] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation. In *International Symposium on High Performance Computer Architecture (HPCA)*. 173–184.

[51] Wei Xu, Sandeep Bhatkar, and R. Sekar. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX Security Symposium (SEC)*. 121–136.

[52] Angeliki Zavou, Vasilis Pappas, Vasileios P Kemerlis, Michalis Polychronakis, Georgios Portokalidis, and Angelos D. Keromytis. 2013. Cloudopsy: An Autopsy of Data Flows in the Cloud. In *International Conference on Human Aspects of Information Security, Privacy, and Trust (HAS)*. 366–375.

[53] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. 2011. TaintExchange: A Generic System for Cross-process and Cross-host Taint Tracking. In *International Workshop on Security (IWSEC)*. 113–128.

[54] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting Sensitive Data Leaks using Application-level Taint Tracking. *ACM SIGOPS Operating Systems Review* 45, 1 (2011), 142–154.