`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

Virtual Execution Environments (VEE), 03/04/2012

Outline

1. Overview
   - Problem statement
   - Contribution
2. Design & Implementation
   - Definitions
   - Design overview
   - Implementation
3. Results & Discussion
   - Performance
   - Use cases
   - Summary

Dynamic data flow tracking (DFT)

What is it?

- **Tagging** and **tracking** “interesting” data as they propagate during program execution
- Extremely popular research topic (also known as information flow tracking)
  - analyzing malware behavior [Portokalidis Eurosys’06]
  - hardening software against zero-day attacks [Bosman RAID’11, Qin MICRO’06, Newsome NDSS’05]
  - detecting and preventing information leaks [Zhu SIGOPS’11, Enck OSDI’10]
  - debugging software misconfigurations [Attariyan OSDI’10]

Related work

Architectural classification

- Integrated into full system emulators and virtual machine monitors [Ho Eurosys'06, Portokalidis Eurosys'06, Myers POPL'99]
- Retrofitted into unmodified binaries using dynamic binary instrumentation (DBI) [Qin MICRO'06]
- Added to source codebases using source-to-source code transformations [Xu USENIX Sec'06]
- Implemented in hardware [Venkataramani HPCA'08, Crandall MICRO'04, Suh ASPLOS'04]

Related work (cont'd)

Issues & limitations

Ad hoc & problem-specific implementations

high overhead, little reusability, limited applicability

Attempts for flexible DFT systems

Versatility comes at a high price

- TaintCheck [Newsome NDSS'05] → 20x overhead even for small utilities
- LIFT [Qin MICRO'06] → no multithreading support
- Minemu [Bosman RAID'11] → only 32-bit binaries
- Dytan [Clause ISSTA'07] → attempts to define a generic and reusable DFT framework, but incurs a slowdown of more than 30x

libdft

Brief overview

- DFT framework in the form of a shared library

Features

- **Fast** → 1.14x – 10x slowdown
- **Reusable** → API for building *custom* DFT-powered tools
- Applicable to **commodity hardware and software** → supports multi-{process, threaded} x86 Linux applications, without requiring any modifications to the binaries or the underlying OS

DFT

Formalisms

- Many aliases
  - Data flow tracking (DFT)
  - Information flow tracking (IFT)
  - Dynamic taint analysis (DTA)
  - ...

Definition

The process of **accurately** tracking the flow of **selected** data throughout the execution of a program or system

DFT (cont'd)

Basic aspects

- DFT is characterized by 3 aspects
  1. **Data sources**: program, or memory locations, where *data of interest* enter the system and subsequently get *tagged*
  2. **Data tracking**: process of *propagating* data tags according to program semantics
  3. **Data sinks**: program, or memory locations, where *checks* for tagged data can be made

Note

We strictly deal with **explicit** data flows

Design goal

Shared library for *customized* DFT

Allow the creation of “meta-tools” that *transparently* employ DFT

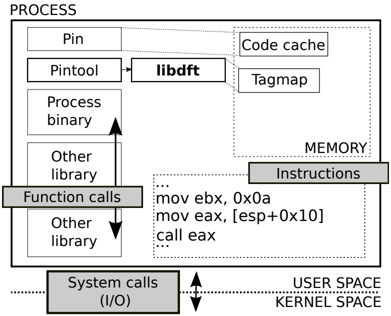


Figure: Putting it altogether: Pin, libdft, process

Usage

libdft in a nutshell

1. Pin loads itself, libdft, and a libdft-enabled tool into the same *address space* with a process (Figure 1)
2. Before commencing or resuming execution, the libdft-tool defines the data sources and sinks by *tapping* arbitrary points of interest
3. User-defined callbacks drive the DFT process by tagging and untagging data, or checking and enforcing data use

Challenges

Achieving low overhead is hard

- Size & structure of the analysis routines (*i.e.,* DFT logic) matters
- Complex analysis code → excessive register spilling
- Certain types of instructions should be avoided altogether (*e.g.,* test-and-branch, `EFLAGS` modifiers)

libdft

Prototype implementation

- libdft has been implemented using Pin v2.9
- Currently supports only x86 Linux binaries
- Consists of three main components (Figure 2)
  1. **Tagmap**
  2. **Tracker**
  3. **I/O interface**
- $\sim$5000 LOC in C/C++

libdft

Architecture

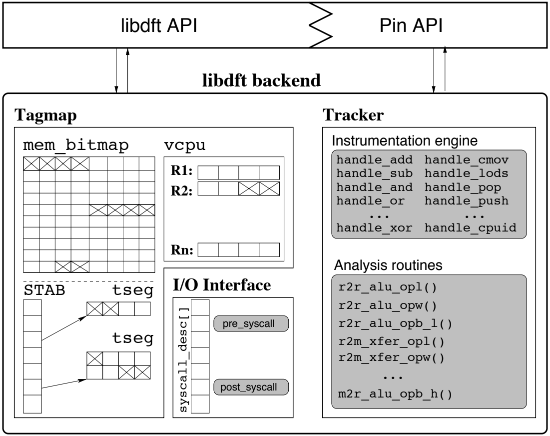


Figure: The architecture of libdft

libdft

Tagmap

- Stores the tags for every process
- Major impact on the overall performance → DFT logic constantly operates on data tags
- Tag format
  - **Tagging granularity** → byte
  - **Tag size** → {1,8}-bit
- Register tags
  - Per thread `vcpu` structure
  - 8 general purpose registers (GPRs)
- Memory tags
  - Per process `mem_bitmap`, or `STAB` and `tseg` structures
  - 1 bit/byte for every byte of addressable memory

libdft

Tracker

- Instruments a program for retrofitting the DFT logic
- **Instrumentation Engine**
  - Invoked *once* for each sequence of instructions
  - Handles the elaborate logic of discovering data dependencies → allows for compact and fast analysis code
  - *Inspects* the instructions of a program
  - *Determines* the analysis routines that should be injected before each instruction
  - Allows for customization (libdft API)
- **Analysis Routines**
  - Invoked *every time* a specific instruction is executed
  - Contain code that implements the DFT logic
  - Clear, assert, and propagate tags

libdft

I/O Interface

- Handles the kernel ↔ process data
- `pre_syscall/post_syscall` → **instrumentation stubs**
- `syscall_desc[]` → syscall **meta-information table**
- The stubs are invoked upon every system call entry/exit
- Allows the user to register callback functions (libdft API)
- The default behavior of the `post_syscall` stub is to untag the data being written/returned by the system call

Advantages

- Enables the *customization* of libdft by using I/O system calls as data sources and sinks arbitrarily
- *Eliminates* tag leaks by considering that some system calls write specific data to user-provided buffers

libdft

Optimizations

`fast_vcpu`	Uses a scratch-register to store a pointer to the `vcpu` structure of each thread
`fast_rep`	Avoids recomputing the effective address (EA) on each repetition in `REP`-prefixed instructions
`huge_tlb`	Uses huge pages for `mem_bitmap` and `STAB` to minimize TLB poisoning
`tagmap_col`	Collapses `tseg` structures that correspond to write-protected memory regions to a single constant segment

Performance evaluation

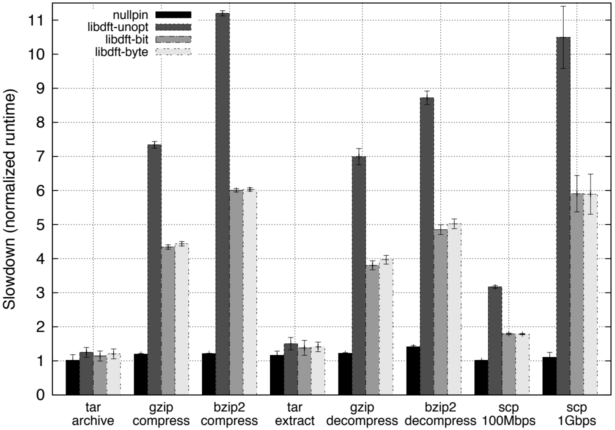
Testbed

- 2 identical hosts
    - 2x 2.66GHz quad core Intel Xeon X5500 CPUs
    - 24GB of RAM
- 4 Pintools
    - **nullpin** → runs a process over Pin
    - **libdft-unopt** → Pin+libdft with no optimizations
    - **libdft-bit** → Pin+libdft with optimizations and bit-sized tags
    - **libdft-byte** → Pin+libdft with optimizations and byte-sized tags
- Debian GNU/Linux v6 (squeeze), kernel version 2.6.32
- Pin v2.9 (build 39586)

GNU command-line utilities, Apache v2.2.16, MySQL v5.1.49, Firefox v3.6.18

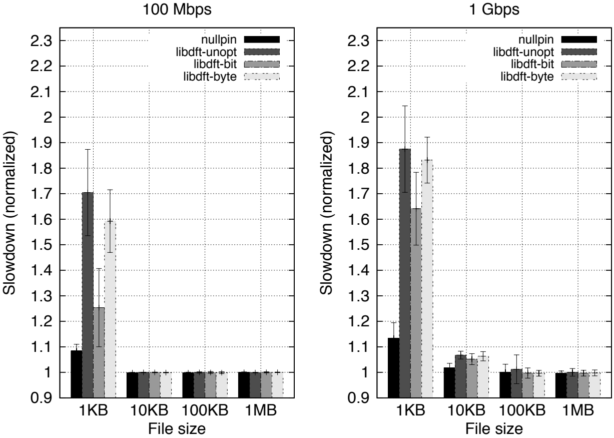
Performance evaluation

Command-line utilities



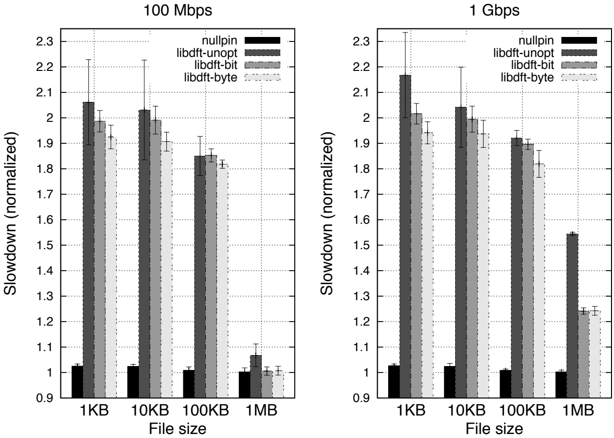
Performance evaluation

Apache web server



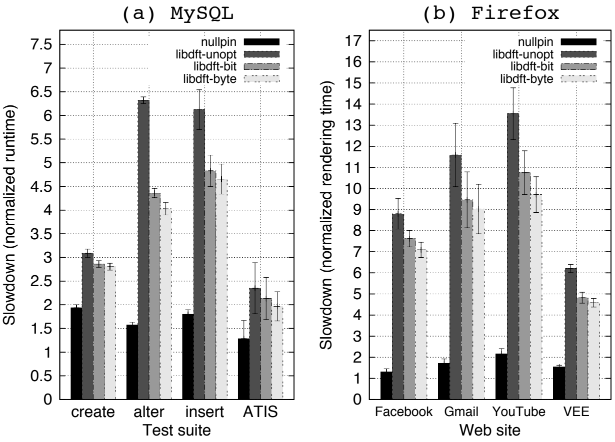
Performance evaluation

Apache web server (cont'd)



Performance evaluation

MySQL RDBMS & Firefox web browser



libdft-DTA

Taint analysis made easy

- libdft offers a small and elegant API for *transparently* incorporating DFT into running applications → can we use it in order to enforce security policies?
- Built a full-fledged DTA tool in ∼ 450 LOC that protects against *code injection* attacks (*e.g.*, stack smashing, heap corruption) *memory overwrite* attacks (*e.g.*,`return-to-libc`, format string) *etc.*
- +7% additional runtime overhead
- Tested with real exploits

Dynamically retrofit DTA capabilities into running applications
→ Binary inline reference monitor

Recap

`libdft`

- **Fast** (highly optimized *Tracker*)
    - *branch-less* tag propagation
    - *single assignment* tagmap updates
    - *inlined* DFT logic
- **Reusable** (API)
    - *customizable* propagation policy
    - assignment of data sources and sinks at *arbitrary* points of interest
- Applicable to **commodity hardware and software**
    - *multiprocess* and *multithreading* support
    - no *modifications* to the binaries or the underlying OS
- `www.cs.columbia.edu/~vpk/research/libdft/`

Backup slides

DFT

Explicit vs. implicit data flows

```
unsigned char csum = 0;

bcount = read(fd, data, 1024);
while(bcount-- > 0)
      csum ^= *data++;

 write(fd, &csum, 1);
```

(a) Data flow dependency

```
int authorized = 0;

bcount = read(fd, pass, 12);
MD5(pass, 12, phash);
if (strcmp(phash, stored_hash) == 0)
    authorized = 1;
return authorized;
```

(b) Control flow dependecy

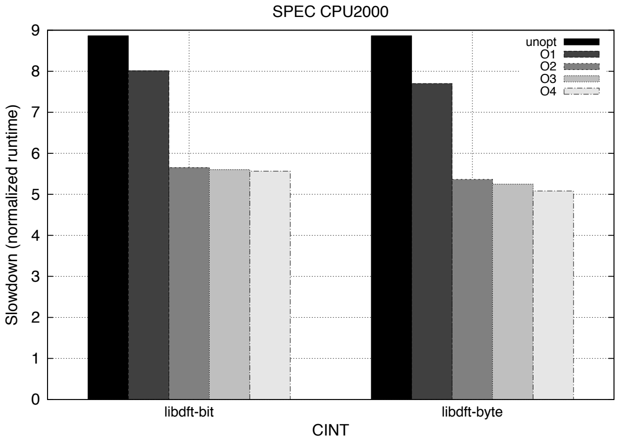
Figure: Examples of code with explicit and implicit data dependencies

Pin DBI

- libdft relies on Pin [Luk PLDI'05] for *instrumenting* and *analyzing* the target process
- **Instrumentation** → *what* analysis routines should be inserted *where*
- **Analysis routines** → code that is dynamically injected into the application and *augments* its execution
- Pin uses a *JIT* compiler for combining the original code, libdft, and the code of a libdft-tool
- "Jitted" code is placed into a *code cache* for avoiding re-translation

Performance evaluation

SPEC CPU2000 benchmark



References I

- M. Attariyan and J. Flinn.
  Automating configuration troubleshooting with dynamic information flow analysis.
  In *Proc. of the* 9th *OSDI*, pages 237–250, 2010.
- E. Bosman, A. Slowinska, and H. Bos.
  Minemu: The World's Fastest Taint Tracker.
  In *Proc. of the 14th RAID*, pages 1–20, 2011.
- J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum.
  Understanding Data Lifetime via Whole System Simulation.
  In *Proc. of the* 13th *USENIX Security*, pages 321–336, 2004.

References II

- J. Clause, W. Li, and A. Orso.
  Dytan: A Generic Dynamic Taint Analysis Framework.
  In *Proc. of the 2007 ISSTA*, pages 196–206.
- J. R. Crandall and F. T. Chong.
  Minos: Control Data Attack Prevention Orthogonal to Memory Model.
  In *Proc. of the* 37th *MICRO*, pages 221–232, 2004.
- 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* 9th *OSDI*, pages 393–407, 2010.

References III

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.

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.

References IV

- 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.
- A. C. Myers.
  JFlow: Practical Mostly-Static Information Flow Control.
  In *Proc. of the 26th POPL*, pages 228–241, 1999.
- J. Newsome and D. Song.
  Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software.
  In *Proc. of the 12th NDSS*, 2005.

References V

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

References VI

- 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.
- 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.
- 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.

References VII

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.