Taint-Exchange: a Generic System for Cross-process and Cross-host Taint Tracking

Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis

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

Abstract. Dynamic taint analysis (DTA) has been heavily used by security researchers for various tasks, including detecting unknown exploits, analyzing malware, preventing information leaks, and many more. Recently, it has been also utilized to track data across processes and hosts to shed light on the interaction of distributed components, but also for security purposes. This paper presents Taint-Exchange, a *generic* cross-process and cross-host taint tracking framework. Our goal is to provide researchers with a valuable tool for rapidly developing prototypes that utilize cross-host taint tracking. Taint-Exchange builds on the *libdft* open source data flow tracking framework for processes, so unlike previous work it does not require extensive maintenance and setup. It intercepts I/O related system calls to transparently multiplex fine-grained taint information into existing communication channels, like sockets and pipes. We evaluate Taint-Exchange using the popular *lmbench* suite, and show that it incurs only moderate overhead.

1 Introduction

Dynamic taint analysis (DTA) has been a prominent technique in the computer security domain, used independently or frequently complementing other systems [9,23,21,19,20,27,26,7], while researchers seem to continuously find new applications for it, many times extending to other domains [12,29]. Originally, taint tracking systems enabled the tracking of marked or "tainted" data throughout the execution of a single process [21], or an entire host in the case of virtual machine (VM)- and emulator-based systems [13,20]. The latter enabled researchers to track the interactions between processes running within a virtual machine.

However, as taint tracking is applied on more domains, like the visualization of information flow among the components of a system [17,28] and the automatic troubleshooting of application misconfigurations [1], systems that can also propagate taint between different hosts over the network have been also developed. Existing cross-application and cross-host taint propagation systems frequently make use of VMs and emulators [17,28], incurring unnecessary overhead and requiring extensive maintenance and setup. Other implementations are very problem-specific, requiring extensive modifications for reuse by the research community to solve new problems.

This paper presents a *generic* cross-process and cross-host taint tracking framework, called Taint-Exchange. Our system, builds on the *libdft* open-source data flow tracking (DFT) framework [14], which performs taint tracking on unmodified binary processes using Intel's Pin dynamic binary instrumentation framework [15]. We have extended *libdft* to enable *transfer* of taint information for data exchanged between hosts through network sockets, and between processes using pipes and unix sockets. Taint information is transparently *multiplexed* with user data through the same channel (*i.e.,* socket or pipe), allowing us to mark individual bytes of the communicating data as tainted. Additionally, users have the flexibility to specify which communication channels will propagate or receive taint information. For instance, a socket from *HOST A* can contain fine-grained taint information, while a socket from *HOST B* may not contain detailed taint transfer information, and all data arriving can be considered as tainted. Similarly, users can also configure Taint-Exchange to treat certain files as tainted. Currently, entire files can be identified as a source of tainted data.

Most real-world services consist of multiple applications exchanging data, that in many cases run on different hosts, *e.g.,* Web services. Taint-Exchange can be a valuable asset in such a setting, providing transparent propagation of taint information, along with the actual data, and establishing accurate cross-system information flow monitoring of *interesting* data. Taint-Exchange could find many applications in the system security field. For example, in tracking and protecting privacy-sensitive information as it flows throughout a multi-application environment (*e.g.,* from a database to a web server, and even to a browser). In such a scenario, the data marked with a "sensitive" tag, will maintain their taint-tag throughout their lifetime, and depending on the policies of the system, Taint-Exchange can be configured to raise an alert or even restrict their use on a security-sensitive operation, *e.g.,* their transfer to another host. In a different scenario, a Taint-Exchange-enabled system could also help improve the security of Web applications by tracking unsafe user data, and limiting their use in JavaScript and SQL scripts to protect buggy applications from XSS and SQL-injection attacks.

Taint-Exchange, along with libdft, provides a stable and reusable cross-host taint tracking platform that can promote new research and expedite the development of research prototypes. The main contributions of this paper are summarized in the following:

- We designed and implemented a *reusable* cross-process and cross-host taint tracking framework. Taint-Exchange is based on *libdft* [14], a customizable DFT framework that offers an extensive API for creating tools
- Taint-Exchange operates transparently on unmodified x86 Linux binaries, allowing real-world legacy applications to take advantage of our framework transparently
- We offer flexible configuration of taint sources, as well as allowing mixing our own fine-grained taint transferring sockets with ordinary sockets. For example, many security-oriented DTA implementations [19] do not support configurable taint sources, and mark all incoming network as tainted

– We improved on inter-process taint tracking over previous system-wide tracking systems (e.g. Minos [9], TaintBochs [7], Rakscha [10], RIFLE [24]), which are based on slow full-system emulators (e.g. Xen [2], QEMU [3], Bochs [4]), by enabling cross-host and cross-process tracking on the communication channels that matter to the target applications, rather than overloading every operation in the entire system with unnecessary heavyweight taint tracking operations
– We evaluate the overhead imposed by Taint-Exchange, and show that it incurs minimal overhead over the baseline tool *libdft*

The rest of the paper is organized as follows. Section 2 introduces the concept of dynamic taint tracking and presents the most important implementations in this research area. In Sect. 3 we present our protocol and our system. In Sect. 4, we highlight the implementation choices made when we built our system. In Sect. 5, we evaluate our system. We discuss future work in Sect. 6, and finally, our conclusions follow in Sect. 7.

2 Background and Related Work

2.1 Dynamic Taint Tracking

Dynamic taint tracking is the mechanism of monitoring the flow of tainted data, at runtime, within an instance of a software application (process) or a system, after "recognizing" the *data of interest* according to a predefined taint configuration, and associating it with metadata, usually referred to as *taint tags*. Therefore, most dynamic taint analysis implementations can be described by three elements the *taint sources*, the *propagation policy* and the *taint sinks*. Regarding taint-tags, in most cases one bit of taint is sufficient, but there are situations where multiple bits are useful. For instance, to distinguish between multiple input sources or to distinguish between trust levels.

Dynamic taint tracking is not a new concept. One of its first practical instantiations was employed in detecting and defending against software attacks [19], while it has found many more applications since then. Currently, dynamic tracking approaches range from *per-process* taint tracking [6,8,14,19,21,25,29], to *whole-system* tracking [9,10,20,27] using emulation environments and hardware extensions.

2.2 Single-process Taint Tracking

Most application-level taint tracking tools, like TaintCheck [19], TaintTrace [6], *libdft* [14], Dytan [8], and LIFT [21] use dynamic binary instrumentation (DBI) frameworks, like PIN [15], StarDBT [5] and Valgrind [18]. While quite effective and useful, as they do not require any modifications to source code or customized hardware, they impose significant impact on the performance, as every instruction needs to be instrumented, and additional storage, usually called *shadow*

memory, is required for storing the tags. As a result, there has been great interest in optimization techniques in order to improve their performance. TaintTrace achieved significantly faster taint-tracking by using more efficient instrumentation based on DynamoRIO, combined with simple static analysis to speed up the taint-tag access. LIFT also achieved significant additional performance benefits by using better static analysis and faster instrumentation techniques.

2.3 Cross-process and Cross-host Taint Tracking

A large body of research has also focused on cross-process or system-wide taint tracking, leading to the creation of many tools [9,10,27,28], mostly based on emulators and hardware extensions to efficiently handle data tracking for an entire operating system (OS). For instance, the whole system emulator QEMU [3] is employed by various solutions that implement DTA [13,20,27], while TaintBochs [7] builds on the Bochs IA-32 emulator. The architecture community attempted to integrate or assist dynamic taint tracking with hardware extensions [9,10,23,24], to alleviate the significant performance impact due to extra tag processing from DBI frameworks and emulators.

While there is much research aiming at intra-process and system-wide DTA implementations, it was not until very recently that interest has risen for efficient cross-host taint propagation systems [1,11,28]. Most of these techniques are more problem-specific, and therefore it would be difficult to adapt the techniques and tools developed for use in other contexts. For instance, DBTaint [11] is targeting taint information flow tracking specifically for databases, whereas ConfAid [1], which is the closest to our design, tackles the problem of discovering a set of possible root causes in configuration files that may be responsible for software misconfigurations. System tomography [17], which also looks into the concept of propagating taint information remotely, builds on the QEMU emulator so it cannot be applied on already deployed software and incurs large slowdowns. Finally, Neon [28] also requires modifications in the underlying system to perform dynamic taint tracking. It uses a modified NFS server for handling the initial tainting, and utilizes a network-filter for monitoring the tainted packets arriving/leaving the server.

In contrast to previous approaches, that use slow-emulators or VMs to perform system-wide taint tracking, in Taint-Exchange we use an already available and fast single-process taint-tracking framework [14], and extend it to perform fine-grained, cross-process, and cross-host transfer of taint information. *Although our design was inspired by prior works, it addresses different challenges, is more general, and completely transparent to applications.*

3 Taint-Exchange

We designed and implemented Taint-Exchange based on the *libdft* data flow tracking framework [14], to produce a *generic* system for efficiently performing cross-process and cross-host taint tracking. Nevertheless, Taint-Exchange could

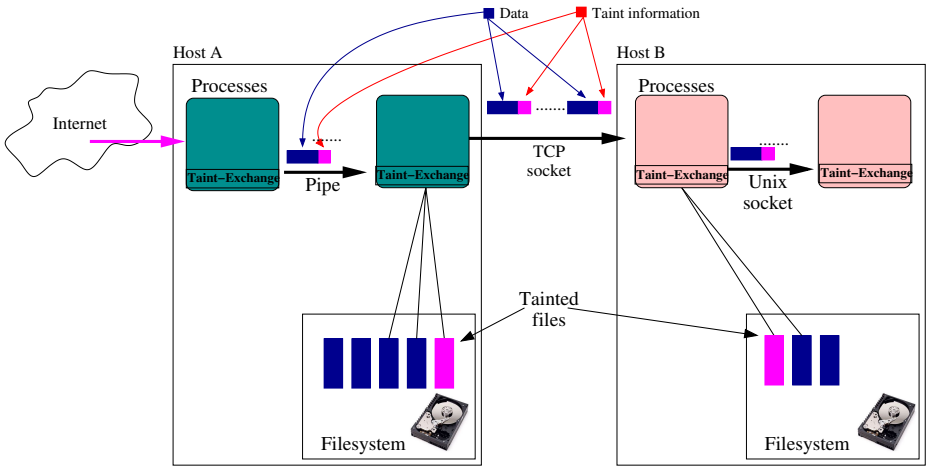

Fig. 1. Taint-Exchange overview. Taint information can be exchanged using network sockets and pipes. Sockets and files can also be configured as taint sources.

be easily retargeted to numerous other taint tracking systems similar to libdft, *e.g.,* TaintCheck [19], TaintTrace [6], LIFT [21], or Dytan [8], as Taint-Exchange is a broadly applicable design. libdft was chosen because it is one of the fastest process-wide taint-tracking frameworks, which performs at least as fast, if not faster than similar systems such as LIFT and Dytan.

3.1 Design Overview

We will present the main aspects of Taint-Exchange, following the three-dimensions described in Sect. 2. Figure 1 shows an overview of Taint-Exchange, the various sources of tainted data that can be configured, and the mechanisms for exchanging taint information between processes and hosts.

Taint sources The taint sources are the "starting points" of the system, where taint is assigned to *data of interest*. Our current framework supports configurable taint sources from the two most common input channels, the file-system and the network. In the current implementation, the user of Taint-Exchange directly interacts with the underlying framework (*i.e.,* both Taint-Exchange and libdft) to define the taint sources, as they each time depend on the problem being tackled. Although configuration is straightforward, we stress that a better user interface for the configuration of the taint sources would improve usability, but this is beyond the scope of this paper.

Configuring the filesystem taint sources is straightforward. A shadow file `taint_config` is maintained for listing all tainted files in the system. The designer has to update it with the taint files, using full-path format, and all data originating from files listed in `taint_config` will be marked as tainted.

In addition, network sockets and pipes can also be among the taint sources, so data arriving from them can be also tagged as tainted. Sockets and pipes can be also configured to receive and transmit detailed taint information regarding the data being exchanged (described later in this section).

A global array (*state_sfd*) is used to keep track of the important "channels" that comprise Taint-Exchange's taint sources. The `open()`, `socket()`, `accept()`, `dup()`, and `close()` system calls are intercepted to update the `state_sfd` array accordingly. The marked "channels" will be the ones monitored for tainted data. Briefly, for read-like calls, such as `read()`, `readv()`, `recv()` etc., this includes the extraction of a taint-header from the received data stream, the reception of the taint information, and the appropriate marking of the received data.

Data tags Currently our system only supports binary tags (tainted or clean data), but this is only a limitation because of the chosen underlying taint tracking system *libdft* [14]. In fact, according to the authors of libdft, future versions will include support for multiple labels/colors for tracked data. In the future, we plan to port our tool to using the lated libdft version to take advantage of the richer data tags. Of course, we expect larger tags to have a larger impact on the performance of data transfers, but it is something that needs to investigated.

Taint propagation There are three cases that we examined for the propagation of taint tags. Firstly, the *intra-propagation* of taint values during the execution of a single process. As we discussed in Sect. 2, this has been thoroughly explored by past work, and there exist many tools [6,8,14,19] for efficiently handling this issue. Generally, all these tools allocate a "shadow storage" for every process to store which data is tainted (*i.e.,* data tags). We will refer to this shadow memory as a `tagmap`. The second case of taint propagation we consider is the *cross-process* propagation of taint tags for the data exchanged between processes. Previous research has mostly addressed this topic by performing system-wide taint tracking using modified VMs and specialized hardware [9,10,20,27], mostly based on emulators and hardware extensions for efficiently handling system-wide tracking of tainted information. The last case we examine is the *cross-host* transfer of tainted tags. Relatively little research has explored this path [1,11,17,28].

For Taint-Exchange, **taint transfer** refers to the propagation of taint information along with the data, when processes on the same host or on different hosts communicate. Our mechanism supports processes that communicate using sockets and pipes. Briefly, the main idea is to monitor the information flow between the taint sources, and intercept the system calls from the `read/receive` and `write/send` families that are used to read from and write to the tainted channels. In the case of data leaving the current process (or host), a **taint-header** is composed and attached to the data, indicating which bytes, if any, are tainted. We will describe the taint-header in detail in Sect. 3.2. On the receiving side, as "extended" data enters the process (or host), and assuming that the source descriptor is among the taint sources described earlier, the taint header

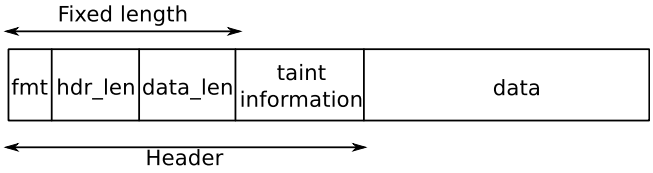

Fig. 2. Taint-Exchange encapsulates data using a header to transparently inject taint information in data transfers.

will be extracted from the received data, and the taint-tag storage structure of the process will be updated accordingly.

Cross-process taint transfer is handled the same way as cross-host taint transfer. The only difference is the source descriptor, which in the case of pipes is the pid or name of the application we are communicating with. IPC through UNIX sockets is very similar to TCP sockets, so the mechanism remains almost entirely the same.

Taint sinks *Taint sinks* refer to the "locations" in the system, where the user needs to perform some assertions on the data. For example, tainted data may not be allowed to be transmitted overa certain socket, or used as program control data (*e.g.,* a function return address). or it should just be logged. Taint sinks are problem-specific, and can be configured by the user. *libdft* offers an extensive API for the users to check for the presence of tainted data on instructions and on system or function calls.

3.2 Taint Headers

Taint Header Structure To multiplex data and taint information, Taint-Exchange prefixes each data transfer with a taint header, which essentially encapsulates the transferred data into new "packet" protocol, following the format shown in Fig. 2. It consists of the following fields:

`fmt` the format version of that taint information
`hdr_len` the length of the header including that taint information
`data_len` the length of the data payload (*i.e.,* the data the application is actually transferring)
`taint information` fine-grained taint information regarding the payload, and following the format specified by `fmt`
`data` the data payload

Composing the Taint Header A taint header is created when `write-like` system calls, such as `write()`, `writev()`, `send()` etc., are executed and the destination descriptor of the system call is among the ones configured to transfer taint. The process' `tagmap` is referenced to determine which parts of the outgoing data message is tainted. Depending on the number of tainted bytes and

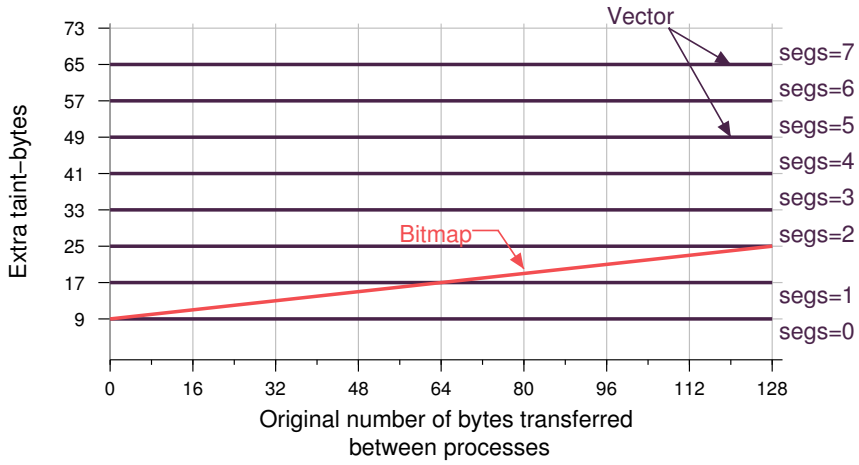

Fig. 3. Space overhead of the different Taint-Exchange taint information transfer formats. For both formats, the fixed header size is 9 bytes. The overhead of the bitmap format is linear with the number of bytes transferred, while with the vector protocol it depends solely on the number of tainted data segments.

their distribution, Taint-Exchange determines which format to use to encode the taint information for the data. We support two formats for encoding taint information. The first, is a *bitmap* which contains one bit for every byte of data being transferred, and the second using a *vector* for each segment of data that is tainted. For example, if bytes 5 through 15 are tainted, the vector describing this segment is [5, 10]. That is, there is a segment of tainted data starting at offset 5 of the data and lasting for 10 bytes. The space overhead of these two formats is drawn in Fig. 3. We see that depending on the number of tainted segments a different format is preferable. Usually, for large continuous areas of tainted bytes, a vector proves to be a more efficient choice, whereas for sparsely tainted bytes the bitmap is preferable.

4 Implementation

4.1 The libdft Data Flow Tracking Framework

Taint-Exchange operates by intercepting and instrumenting the system calls used for inter-process as well as for cross-host communication, while it relies on an already available tool, *libdft* [14], to perform the taint tracking within each process. For this purpose, we instrumented the `socketcall` family of system calls (*i.e.,* `socket()`, and `accept()`), the `dup()` system call, and the `read/receive`-like and `write/send`-like system calls. We also intercept the `open()`, `close()` and `mmap()` system calls for handling files.

For the intra-process dynamic taint tracking we chose *libdft* [14], a dynamic Data Flow Tracking (DFT) framework, designed to transparently perform DFT on binaries. Although our design is independent of the underlying IFT system, for our implementation we chose *libdft* because of its availability, well-defined API, and efficient instrumentation, which makes it one of the fastest process-wide taint-tracking frameworks. *libdft* is used as a shared library offering a user-friendly API for customizing *intra-process taint propagation*, and can be used on unmodified multi-threaded and multiprocess applications. It relies on PIN [15], a dynamic binary instrumentation framework (DBI) from Intel, widely used in the implementation of other DTA tools [8,29].

Similarly to PIN's, the *libdft* API allows both instrumentation and analysis of the target process. In particular, *libdft*'s *I/O interface* component, offers an extensive *system call* level API, enabling instrumentation hooks before (`pre_syscall`) and after (`post_syscall`) every system call, while making use of the underlying DFT services. We have registered analysis callbacks for the "interesting" system calls, which get invoked when these system calls are encountered, to observe the process' communication "channels", and to inject taint information along with the native data (*i.e.,* the data the application is communicating). For the system calls that are not explicitly handled by Taint-Exchange, the default behavior is to clear the tags of the data being read (*i.e.,* the data written in the process' memory). This way, over-tainting is avoided since "uninteresting" system calls, that overwrite program memory with new inputs read from the kernel, are not ignored. It is worth to mention that *libdft* does not suffer from taint-explosion (*i.e.,* over-tainting data that should not be tagged), because it does not consider control-flow dependencies and it operates in user-space. Control-flow dependencies, kernel data structures, and pointer tainting have been identified as the prominent causes of taint-explosion [22].

4.2 Taint-Exchange Data Structures

The *tagmap* is the taint-tag storage maintained by *libdft*, reflecting the taint-status of the running process' memory and CPU registers for each running thread. Its implementation plays a crucial role in performance and memory overhead. *libdft* supports byte-level memory tagging, which is mapped to a single-bit tag in the process' tagmap, and four 1-bit tags for every 32-bit GPR. *libdft* offers an extensive API for the update of the taint-tags in tagmap (*e.g.,* `tagmap_setb`, `tagmap_getb`, `tagmap_clrb` are handling the taint-tag per byte of addressable memory).

The *state_sfd* is the global array of tainted descriptors, that designates the important "channels", being monitored for tainted data. It is updated by the system call subset used for handling files and creating/closing sockets, and is indexed by the file (or socket) descriptor. Initially, all elements of the array are empty. The array is updated by the instrumented versions of `open()`, `socket()`, `accept()`, `dup()` and `close()` system calls, and variations of them.

Finally, *taint_config* is the configuration file for filesystem taint sources. The files listed in it should be written in full-path format.

4.3 Filesystem Taint Sources

Currently, our framework supports configurable taint sources from the file-system and the network. The `taint_config` file, lists the files that contain *data of interest*, which should be tainted and tracked throughout the monitored system. This is implemented, by the instrumented `open()` system call, which marks as "tainted" the *state_sfd* elements, that correspond to the files listed in *taint_config*. The descriptors of these files are considered "channels" of incoming tainted data. Therefore, whenever a system call, like `read()`, tries to read data from them the corresponding taint-tags in the *tagmap* structure are updated accordingly.

4.4 Taint Propagation Over the Network

The main purpose of Taint-Exchange is the delivery of taint-tags along with the transferred data in all the tainted "channels". To establish information about the TCP channels the `socket()` and `accept()` system calls are instrumented. For simplicity, in the current implementation, every network connection is considered *capable* of propagating tainted data, but this could be easily limited to work only on specific IPs. When a TCP connection is established, *state_sfd* structure is updated accordingly to add the new socket descriptor to the monitored "channels".

The cross-host taint propagation mechanism is handled by the instrumented versions of `write/send`-like system calls on the sending side, and `read/receive`-like system call on the receiving side. When the sender transmits data by invoking a `write()` (or an equivalent) system call, Taint-Exchange constructs the corresponding taint-header according to the relevant taint-tags as reflected in the *tagmap* of the sending process and attaches it to the original data. The receiving side, will invoke an instrumented `read()` call (or an equivalent) to process the "extended data", and update the process' *tagmap* with the taint-tags corresponding to the received data.

4.5 Cross-process Taint Propagation

Interprocess communication can happen through unix sockets, TCP/UDP sockets, pipes, and shared memory. If the processes are communicating via sockets or pipes, taint tags can propagate between communicating processes in the same manner we described in the previous section. The main difference is the source descriptor, which in the case of pipes is linked to the pid or name of the application the process is communicating with. IPC through UNIX sockets is very similar to TCP sockets, so the mechanism remains almost entirely the same. We are currently not handling data exchanged through shared memory segments.

5 Evaluation

The aim of this section is to demonstrate the communication overhead of Taint-Exchange, when transparently passing taint information, along with real data,

across the network. To assess the impact imposed by Taint-Exchange, we performed several micro benchmarks using utilities from the *lmbench* [16] Linux performance analysis suite. During the tests we only used the bitmap format to represent taint information as the overhead of the vector format can vary significantly depending on the application scenario.

Our testbed consisted of two identical hosts, equipped with two 2.66GHz 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 39599). When conducting our experiments, the hosts were idle with no other user processes running under taint-tracking apart from the evaluation suites.

Since Taint-Exchange intercepts socket connection calls to inject the additional taint information, we used lmbench's bandwidth benchmark *bw_tcp* to measuring the impact of our approach when moving the "extended" data over the network. bw_tcp measures TCP bandwidth by creating two processes, a server and a client, that are moving data over a TCP/IP connection. We repeated our tests with data of different sizes (*i.e.,* 64, 128, 256, 512, 1024 and 1047 bytes), and against three different scenarios: (a) using a simple pintool, *null_tool*, which uses minimum PIN instrumentation to add callbacks to system calls without further employing any form of analysis. We developed this as the base case, to establish the lower bound of our instrumentation and analysis overhead as imposed by PIN's runtime environment alone. (b) *libdft-dta*, a tool based on *libdft* to employ basic dynamic taint analysis. We used this tool to achieve an estimation of the overhead imposed by *libdft*. (c) Taint-Exchange.

We repeated the measurements 10 times and calculated the mean and standard deviation of the output. The results are presented in Figure 4. We see that there is an obvious impact on the throughput of TCP sockets, which becomes more severe as the size of the sent data increases. As expected, Taint-Exchange has the largest impact of the three scenarios as the number of data sent every time is more than the other applications. For example, in the case of the 64 bytes buffer sent, the space overhead will be 17 bytes (as described in Section 3.2).

When running our tests we noticed that there was an instability in the measurements, as size of the buffer increased, and especially with the libdft-dta tool. We assume that maybe the measurements are affected also by the instruction instrumentation that libdft-dta employs. Unfortunately, we have not yet determined the exact reasons for this instability as we are not fully aware of lmbench's internal workings. The 1437B buffer in our experiments is a default of the lmbench benchmark, and has to do with the maximum number of (payload) bytes that can be transmitted within a single Ethernet frame. This also explains the oscillation in performance, as the taint information can no longer fit within the same Ethernet frame as the data. We should note that Pin itself introduces significant overhead with small buffer sizes like the ones used by the *lmbench* suite (Fig. 4), reducing throughput by 10x-20x compared with native execution. On the other hand, when using large, 10MB buffers (*i.e.,* the largest buffer measured by *lmbench*), Pin does incur any observable overhead on throughput.

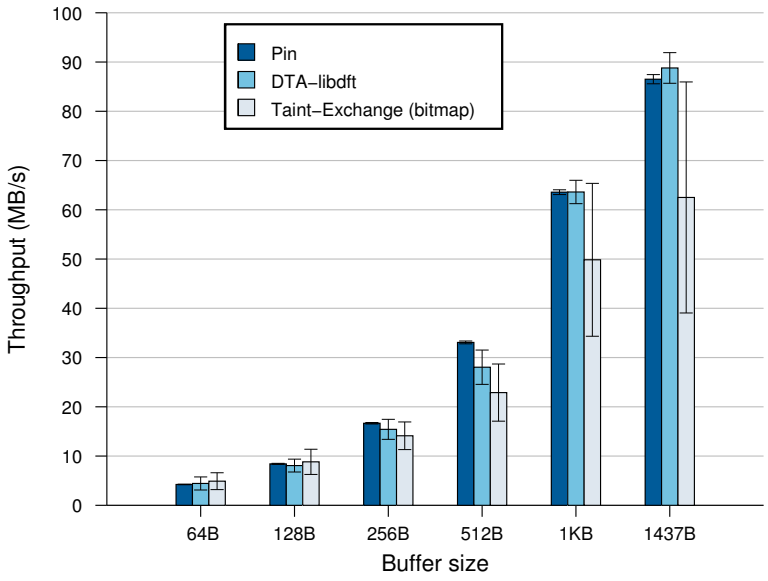

Fig. 4. TCP socket throughput measured with *lmbench* for various buffer sizes. We draw the mean and standard deviation. Note that the effect of Pin with the buffers drawn here is quite pronounced, reducing throughput by 10x-20x compared with native execution. In contrast, it does not incur any observable overhead with large, 10MB buffers.

Since the implementation of Taint-Exchange is mostly based on the instrumentation of system calls, we also employed the *lat_syscall* benchmark to measure the latency impact of the three implementations . We used lat_syscall with the `open`, `read` and `write` system calls, in order to show how these operations are affected. We chose these three system calls as they represent the calls that were affected the most by Taint-Exchange. More specifically, `open()` is handling the initial configuration of tainted "channels" from the file-system, while the `read` and `write` system calls are the ones handling the movement of the tainted information along. In the performed tests, *lat_syscall read* measures how long it takes to read one byte from /dev/zero, whereas *lat_syscall write* measures how long it takes to write one byte to /dev/null. *lat_syscall_open* measures how long it takes to open and then close a file. The results are presented in Figure 5. The conditions during these measurements were the same as with bw_tcp, regarding repetitions and the calculation of the mean and standard deviation from the original measurements.

The observed overhead is attributed to the overhead of PIN for the dynamic instrumentation analysis of the process, as well as the overhead inserted by *libdft* performing the taint-tracking. The additional overhead imposed by Taint-

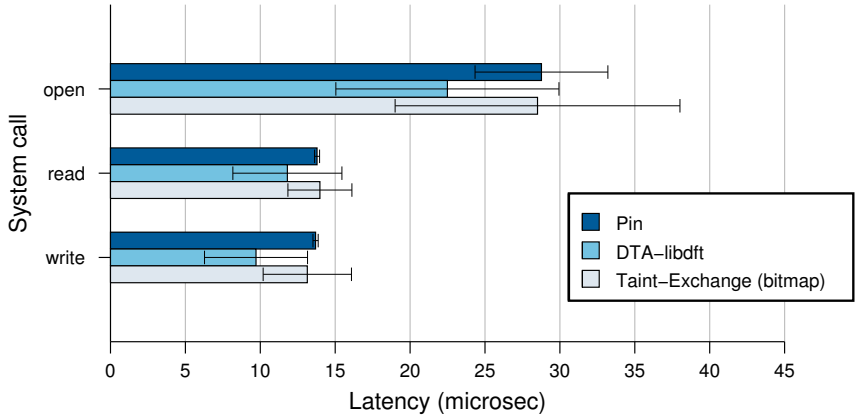

Fig. 5. System call latency measured with *lmbench*. Note that Pin, as probably most DBI frameworks, greatly affects system call latency (approximately 20x slower than native).

Exchange, apart from the obvious reasons such as the instrumentation of these system calls for handling the taint-headers and the continuous update the taint data structures, is also implementation-specific. For instance, in our current implementation every `write()` system call performed by the application results in an additional write being performed to inject the taint-header. Similarly, the instrumented version of the `read()` system call is reading the "extended" data in three pieces, inevitably imposing the overhead seen in Figure 5. Note that Pin, as probably most DBI frameworks, greatly affects system call latency (approximately 20x slower than native) because it receives control before and after every call. We attribute Pin's overhead on throughput with small buffer sizes, to the general increase in system call overhead.

6 Future Work

In this paper, we presented a preliminary implementation of Taint-Exchange, our approach for handling cross-application and cross-host transfer of tainted information. There are some obvious extensions to the work presented in this paper, which we plan to address in a next version of Taint-Exchange. In the current implementation, every TCP socket is by default considered among the important "channels", that participate in the taint-propagation process. We are planning to build a more fine-grained configuration procedure, so that certain IPs can be included (or excluded) from participating into the propagation of the taint-tags over the network.

In addition, we plan to support persistent taint information storage for files, to be able to handle both tainted and untainted data stored in a file. An auxiliary

file per original file could be used to maintain the information on the tainted bytes of the original file. A similar scheme with the one used for passing taint information over the network will be probably used (*e.g.,* bitmap for files with interleaved tainted and clean data, and vectors for files that store tainted data in large blocks). Compression may also be utilized to reduce storage requirements. Coarser-grained tracking can already be performed. For instance, when tainted data are written to a file, taint-exchange can consider the entire file as tainted.

Finally, in order to reduce the overhead of the inserted taint header we think that it is a promising direction to explore the use of TCP optional headers to pass the taint information. This option would not only help us trivially implement communication between a native and a taint-exchange application, but it could also potentially improve the overall performance of our proposed mechanism.

7 Conclusion

We presented Taint-Exchange, a *generic* cross-host and cross-process taint tracking framework. Taint-Exchange enables the transfer of fine-grained taint information across processes and the network. It does so by intercepting I/O system calls to transparently inject and extract information regarding the *taintness* of every byte transferred between processes running under Taint-Exchange. It also provides a flexible mechanism for easily customizing the sources of tainted data, be it a network socket, a file, or an IPC mechanism like a pipe or UNIX socket. Our evaluation of Taint-Exchange shows, as expected, that I/O is affected because of the additional data being sent, and the utilization of the same channel to do so. Nonetheless, we believe that the overhead is small, specially when compared with the high overheads imposed by the various dynamic taint tracking systems.

Acknowledgements This work was supported by the US Air Force and the National Science Foundation through Contract AFRL-FA8650-10-C-7024 and Grant CNS-09-14845, respectively, with additional support by 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, the Air Force, the NSF, Google or Intel. We are also grateful to our shepherd, William Enck, for his assistance in preparing the camera ready version of this paper.

References

1. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: Proceedings of the 9^{th} Symposium on Operating Systems Design and Implementation (OSDI). pp. 1–11 (2010)
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the 19^{th} Symposium on Operating Systems Principles (SOSP). pp. 164–177 (2003)

3. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the USENIX Annual Technical Conference. pp. 41–46 (April 2005)
4. Bochs: The cross platform IA-32 emulator. `http://bochs.sourceforge.net` (2001)
5. Borin, E., Wang, C., Wu, Y., Araujo, G.: Software-based transparent and comprehensive control-flow error detection. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). pp. 333–345 (2006)
6. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S.: TaintTrace: Efficient flow tracing with dynamic binary rewriting. In: Proceedings of the IEEE Symposium on Computers and Communications (ISCC). pp. 749–754 (2006)
7. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding Data Lifetime via Whole System Simulation. In: Proceedings of the 13^{th} USENIX Security Symposium. pp. 321–336 (2004)
8. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA). pp. 196–206 (2007)
9. Crandall, J.R., Chong, F.T.: Minos: Control data attack prevention orthogonal to memory model. In: Proceedings of the 37^{th} Annual International Symposium on Microarchitecture. pp. 221–232 (2004)
10. Dalton, M., Kannan, H., Kozyrakis, C.: Real-world buffer overflow protection for userspace & kernelspace. In: Proceedings of the 17^{th} USENIX Security Symposium. pp. 395–410 (2008)
11. Davis, B., Chen, H.: DBTaint: cross-application information flow tracking via databases. In: Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps) (2010)
12. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9^{th} Symposium on Operating Systems Design and Implementation (OSDI). pp. 393–407 (2010)
13. Ho, A., Fetterman, M., Warfield, C.C.A., Hand, S.: Practical taint-based protection using demand emulation. In: Proceedings of the 1^{st} European Conference on Computer Systems (EuroSys). pp. 29–41 (2006)
14. Kemerlis, V.P.: libdft. `http://www.cs.columbia.edu/~vpk/research/libdft/` (2010)
15. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI). pp. 190–200 (2005)
16. McVoy, L., Staelin, C.: lmbench. `http://lmbench.sourceforge.net/` (2005)
17. Mysore, S., Mazloom, B., Agrawal, B., Sherwood, T.: Understanding and visualizing full systems with data flow tomography. In: Proceedings of the 13^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 211–221 (2008)
18. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI). pp. 89–100 (2007)
19. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12^{th} Annual Network and Distributed System Security Symposium (NDSS) (2005)
20. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks. In: Proceedings of the 1^{st} European Conference on Computer Systems (EuroSys). pp. 15–27 (2006)

21. Qin, F., Wang, C., Li, Z., Kim, H.s., Zhou, Y., Wu, Y.: LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In: Proceedings of the 39^{th} Annual International Symposium on Microarchitecture. pp. 135–148 (2006)
22. Slowinska, A., Bos, H.: Pointless tainting? Evaluating the practicality of pointer tainting. In: Proceedings of EuroSys 2009. Nuremberg, Germany (March-April 2009)
23. Suh, G.E., Lee, J., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of the 11^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 85–96 (2004)
24. Vachharajani, N., Bridges, M.J., Chang, J., Rangan, R., Ottoni, G., Blome, J.A., Reis, G.A., Vachharajani, M., August, D.I.: RIFLE: An architectural framework for user-centric information-flow security. In: Proceedings of the 37^{th} International Symposium on Microarchitecture (MICRO). pp. 243–254 (2004)
25. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proceedings of the IEEE Symposium on Security and Privacy. pp. 497–512 (2010)
26. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: Proceedings of the 15^{th} USENIX Security Symposium (2006)
27. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14^{th} Conference on Computer and Communications Security (CCS). pp. 116–127 (2007)
28. Zhang, Q., McCullough, J., Ma, J., Schear, N., Vrable, M., Vahdat, A., Snoeren, A.C., Voelker, G.M., Savage, S.: Neon: system support for derived data management. In: Proceedings of the 6^{th} International Conference on Virtual Execution Environments (VEE). pp. 63–74 (2010)
29. Zhu, D., Jung, J., Song, D., Kohno, T., Wetherall, D.: TaintEraser: protecting sensitive data leaks using application-level taint tracking. SIGOPS Operating Systems Review 45, 142–154 (February 2011)