2009 International Conference on Availability, Reliability and Security

Capturing Information Flow with Concatenated Dynamic Taint Analysis

Hyung Chan Kim Angelos D. Keromytis
Dept. of Computer Science
Columbia University
New York, NY

Michael Covington Ravi Sahita
Intel Corporation
Portland, OR

Abstract

Dynamic taint analysis (DTA) is a technique used for tracking information flow by propagating taint propagation across memory locations during program execution. Most implementations of DTA are based on dynamic binary instrumentation (DBI) frameworks or whole-system emulators/virtual machine monitors. The boundary of information tracking with DBI frameworks is a single process, while system emulators can cover a host, including the OS. Using system emulators, it may be possible to consider taint propagation across multiple processes executing locally, within the emulator. However, there is an increasing need for tracking information flow across single-system boundaries and across the whole enterprise.

We describe a proof-of-concept architecture for tracking multiple mixed-information flows among several processes across a distributed enterprise. Our DTA tool is based on PIN, a DBI framework by Intel, and the concatenated DTA processing is realized with per-host flow managers. We have tested our prototype with typical enterprise applications. As a motivating example, we track information leakage due to a SQL injection attack from a web-based database server query. Our work is of an exploratory nature, aiming to expose our early findings and identify areas where additional research is needed in improving usability and performance.

1 Introduction

In recent years, security problems related with the flow of information have been increasing. Managers of information systems need to worry about information leakage by web attacks (such as SQL injection and cross-site scripting) or insider activity (whether inadvertent or malicious), among other threats. Especially for web applications, information leakage vulnerabilities are currently considered a top concern [1].

Dynamic taint analysis (DTA) is a technique for tracking information flow within an instance of a software application (process) or a host system. It can be used to detect 0-day (previously unknown) attacks or information leakage, depending on how it is used. A DTA tool tracks information flow via supervising execution of program instructions. Such instrumentation granularity supports substantial coverage of program execution. Most DTA implementations are based on dynamic binary instrumentation (DBI) frameworks, whole system emulators, or virtual machine monitors. Therefore, the boundary of information tracking with DBI frameworks is a single process, while with system emulators it is a single system (including the guest operating system kernel). In the latter case, it may be possible to consider information flow among multiple processes that are executing within the emulated environment. However, there have been no efforts to concatenate DTA processing and tracking across host boundaries to see how information flows among networked systems within an enterprise.

In this paper, we present an experimental framework to realize concatenated DTA processing between processes which may be located in different host systems. Our framework consists of (1) a DTA tool (*SeeC*) and (2) a per-host flow manager.

As with previous DTA work, our DTA implementation dynamically tracks data flow dependencies at the machine instruction level. Our DTA tool is built on a DBI framework so that it can be attached to an existing (binary compiled) software without recompilation, and without requiring source code availability. Moreover, our tool supports 4-byte labels (*i.e., colored tainting*), thereby allowing us to track mixed/multiple information flows simultaneously.

To track information flow between processes, we instrument inter-process communication (IPC) facilities in each monitored process. Our approach is to place *information flow managers* on participating hosts. The managers keep track of their remote peers, and relay inter-process taint information. Currently, our framework supports concatenated DTA processing with TCP channels between DTA-monitored processes.

As a motivating scenario, we describe an experiment us-

978-0-7695-3564-7/09 $25.00 © 2009 IEEE
DOI 10.1109/ARES.2009.56

ing a typical website configuration composed of an Apache web server and a MySQL database server. Setting up an SQL injection attack that triggers information leakage flow from resources in the database server via the web server, we can successfully identify the leakage with our framework. While this is a single, and rather straightforward application of concatenated DTA, we believe that it demonstrates the desirability of such a mechanism. We identify limitations with our current approach and opportunities for further research. We view our work as proof-of-concept, aiming to motivate further work in this space.

Paper Organization Section 2 presents our DTA implementation based on the PIN DBI framework. Section 3 describes ourframework for concatenated DTA processing. Section 4 shows an experiment to capture information leakage by an SQL injection attack. We discuss related work in Section 5.

2 Single-Process DTA

To track information flow of a single program, we have implemented a dynamic taint analysis tool (SeeC) which monitors data flow in a program instance. This section describes our design and implementation of SeeC.

2.1 Information Flow Model

A running program instance (process) is comprised of a sequential execution of machine instructions. When an instruction is executed, data are involved in many cases as the instruction format may take one or more source and/or destination operands. At a single instruction execution, data bytes are copied from one location (register or memory) to another (or the same) location, as specified by the operands. Some instructions may perform data transformations instead of just copying. In both cases, dynamic dependencies exist among memory locations and the data stored therein. Basically, we identify *information flow* from these actions of data copy or transformation: *i.e.,* information flows from memory source(s) to memory destination(s).

There is another type of dynamic dependency, based on program control structure (indirect information flow). For example, in the following code, the two variables a and b have a dependency, as the value of variable b is indirectly decided by the value of a.

```
if (a == 1)
  b = 1;
else if (a > 2)
  b = 100;
```

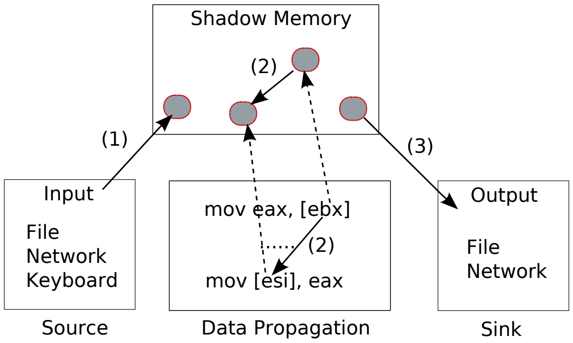


Figure 1. Dynamic taint analysis: (1) **initial marking,** (2) **propagation, and** (3) **assertion**

Our implementation, described next, currently does not consider such control dependencies. Tracking indirect information flow reveals considerably many more dependencies without a commensurate increase in accuracy and relevance; stated another way, tracking indirect information flow often leads to significant "noise" by exposing many more potential data dependencies.

2.2 Design and Implementation

The basic architecture, shown in Figure 1, keeps track of the association between a taint tag in *shadow memory* and memory/registers handled by program instructions. For example, if an instruction causes information to flow directly (*e.g.,* via memory copying) or indirectly (*e.g.,* as part of an arithmetic operation that uses a tainted memory location as an operand) from one memory location to another, our tool updates the taint tags of the corresponding locations in the shadow memory.

The implementation of a DTA tool can be realized with two different view points: (1) whole-system approach [16, 18] based on virtual machine monitors [5] or system emulators [6, 7], and (2) process-oriented approach [13, 14] based on dynamic binary instrumentation (DBI) frameworks [2, 3, 4]. Our implementation is based on PIN, a DBI framework developed by Intel [2], targeting process-based analysis. PIN supports rich APIs for manipulating processes at runtime.

2.2.1 Shadow Memory for Tag Management

Information flow tracking with DTA involves shadow memory that reflects the taint status of a specific memory area or the registers [1] that are used during the execution of native applications. Depending on usage purposes, the map-

[1] For the registers, we currently consider the 8 general registers of the x86 architecture.

ping granularity between shadow memory and application memory/registers can be different. In some previous work, each bit of process memory has a corresponding unit size of shadow memory (tag) [29, 30]. In our tool, we adopt 1-byte precision to improve performance and reduce memory requirements: each byte of application memory is mapped to a unit in shadow memory. A change of any bit in a byte results in tainting the tag for that byte as a whole.

SeeC supports two different unit sizes in shadow memory: 1 byte in application memory can be associated with a 1-bit or a 4-byte[2] unit in the shadow memory. It is a trade off because using 1-bit unit size tag results in a smaller shadow memory footprint. However, a 4-byte unit size enables a tag to retain more information. For information tracking purposes, we use 4-byte tags used as a bitmap. Our tool can thus record 32 values for any process memory byte, realizing *colored tainting* (track the combination of different data). The current version of SeeC manages the shadow memory with a page-table-like structure, allowing us to scale memory requirements with the actual process address space in use.

2.2.2 Tag Propagation

To capture information flow dependent on copy and transformational dependencies, SeeC propagates taint markings according to the following *propagation* and *clearance policies*:

- If at least one source operand is tainted, then the destination operand(s) should be also tainted.
- If the all inputs to an operation are clear, the destination operand(s) should be also cleared.

According to the above policies, in a single assigment instance $m = x$ (*e.g., mov*) the tag would be propagated as $tag(m) = tag(x)$ ($tag(x)$ means the tag value of the memory or register associated with variable x). As the value of m is overwritten by x, it also means that if $tag(x)$ is clear, $tag(m)$ will also be cleared. By simple induction, a sequence of assignments causes transitive propagation thereby realizing flow tracking.

When two or more input operands are involved, *e.g.,* $m = x + y$, the tag value of the destination operand is calculated as $tag(m) = tag(x)|tag(y)$, where | is the bitwise-OR operation. For 4-byte tags, all the bit-fields of the two source operands are preserved at the destination tags. This is an important feature as some instructions that perform transformations may involve *mixed flow of information*. For example, we should identify the mixed flow with the following code snippet.

```
        result[i] = buf1[i] + buf2[i]
```

In the code, `buf1` and `buf2` may come from different data sources. As SeeC supports colored tainting, we can identify both sources referring to the tag associated with the buffer named `result`.

In the x86 architecture, there are a number of other issues to be considered in addition to the basic propagation policy. SeeC includes *implicit operands* for its tag propagation. For example, the *div* instruction involves the eax and/or edx registers together with explicit operands.

As a rule of thumb, we also consider some special situations that result in constant values. For example, `xor %eax, %eax` always causes `%eax` to be filled with the value 0. In that case, `%eax` should be marked as untainted/uncolored. Other cases involve non-deterministic values read from the host machine, such as *rdtsc* [3]. On the other hand, if such values are deterministic, reading them could lead to implicit flow similar to control flow dependencies. Since we do not track implicit flow, all the above cases result in the destination's tag to be cleared.

2.2.3 Taint Sources

Taint sources are starting points where SeeC initially marks a tag for newly introduced data of concern with a proper "color" value. Each color value is associated with a description of the data source: *e.g.,* it may describe where the data comes from or label some other information about the source. The main source points are system calls where read-like operations are performed to introduce data from outside the process, such as from a file, the network, *etc.* Currently, SeeC allows users to designate their source points of interest for file and network sources as a command line option. For those file and network stream sources, users can define a filename or IP address/port for selective sourcing, or can define a whole range of streams (*e.g.,* data from any file, or any TCP streams). Moreover, a user can specify a certain memory area to be tainted for specific applications. SeeC can apply *regular expressions* to incoming data, to select only part for tracking. For example, we can just mark the URI part of an HTTP request for subsequent tracking through a web server.

2.2.4 Taint Sinks

Taint sinks are data destination points, mostly on write-like system calls, where SeeC performs some assertion or validity checking for outgoing file or network stream data. Normal usage of SeeC at a sink point is to check tag information corresponding to the (buffer) memory of the outgoing data. For example, we can identify the original source information if the tag maintains information about the data source,

[2] The size of 4 bytes corresponds to the integer size in 32-bit architectures.

[3] The rdtsc instruction reads the timestamp counter of the x86 processor.

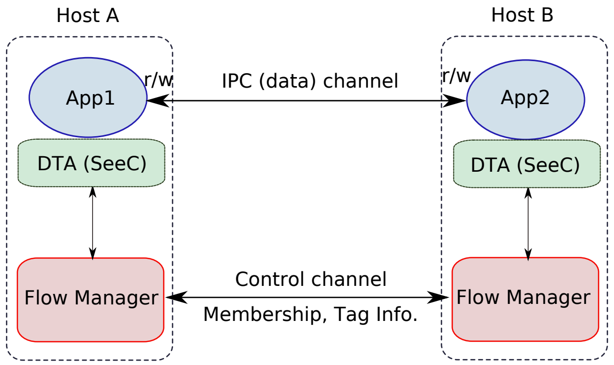


Figure 2. Architecture (peer view)

as the tags are transitively propagated through the process.

3 DTA Across System Boundaries

3.1 System Design and Implementation

In the previous section, we presented an information flow tracking tool that operates within a single application: our DTA tool is attached to a single process to track information flow within its boundary. To realize information flow tracking across multiple applications and even across system boundaries, we need to observe the native data exchanged via inter-process communication (IPC) between interacting applications. Here, we limit our discussion to IPC via TCP connections.

Figure 2 depicts our architecture. The main purpose of our mechanism is to deliver additional tag data when data transfer happens. For example, when a process (App1) sends data to another process (App2) invoking the `write()` or `writev()` system call, the DTA module attached to App1 passes to App2 the corresponding tag data, which may contain source information about the native data currently being transfered. App2 invokes a system call such as `read()` or `readv()` to receive the native data. The SeeC of the receiver side (App2) then appropriately translates the tag data received, and reflects them to its shadow memory. Continuing DTA processing of App2, the information about the data source which comes from App1 can still be maintained and finally be spotted at the sink point(s) of App2. In other words, we concatenate DTA processing of the supervised connections of communicating processes.

A single host may contain multiple processes monitored by DTA tool instances, and a process can make several DTA-supervised connections to its corresponding peers in the same host or in other hosts. To handle multiple DTA processes and connections, we place a *flow manager* in each host. The flow manger handles process membership, session management of DTA for supervised connections, and coordinated delivery of tag data.

3.1.1 Membership and Session Management

A communicating peer may either be participating in information flow tracking under SeeC, or it may just run natively without any instrumentation. We call a process that is supervised with SeeC a *tracking group member*. If the peer process is a member, the source point of one member and the sink point of the other member can be concatenated in terms of DTA processing. Otherwise, SeeC would just record possible information available within the process at the source/sink points. The peer could be another process in the same system or located in a different host.

If a process is launched with SeeC to participate in an information tracking session, SeeC registers its *process id*[4] to the local flow manager, establishing an IPC channel with it. When a TCP communication channel is established with a corresponding process, SeeC queries the peer manager as to the membership status of the peer, so as to decide whether to further perform concatenated DTA processing for the given channel. The flow manager located in the peer host responds to the membership query. As SeeC initiates the query with the established peer's channel information, *i.e., IP address* and *port*, those also need to be registered in the flow manager. It is also possible to communicate with a remote process that is not equipped with SeeC (for which there is no peer flow manager). In that case, the remote process is considered a non-member and concatenated DTA is not performed to that process.

To establish information about the TCP channel session, SeeC instruments `accept()` and `connect()` system calls. When a TCP channel is successfully created, and if the two processes are both members, they each make a membership query about the peer with the given channel information through the interaction of their respective flow managers. Each then stores the peer's information locally so that SeeC can use it for tracking purposes. SeeC also creates a *queue* data structure (`write_q`) for the given channel in the flow manager. After we determine the peer's membership information, the system call returns. The session is managed until the `close()` system call is invoked for the channel.

3.1.2 Tag Data Delivery

To realize concatenated DTA processing, tag data should be delivered with native data to the peer recipient so that the peer process can reflect the tag information in its shadow memory at its source points. The tag recipient should be careful that the received tag data is reflected to the shadow

[4] SeeC and the monitored process operate within the same address space.

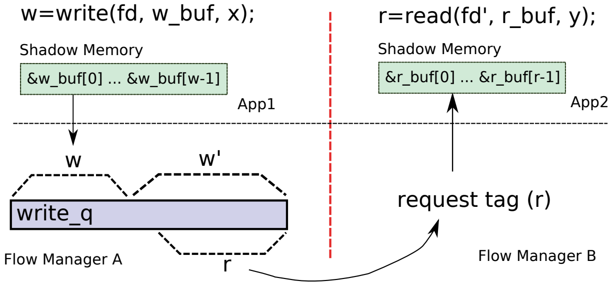


Figure 3. Tag data delivery with a queue

memory area with which the receiving buffer is correctly associated. Unfortunately, dealing with a TCP channel we cannot expect synchronized `write()`–`read()` pairs of system calls in both sides for a given socket descriptor.

However, we can deliver tag data for the successfully transferred native data by using a simple FIFO queue structure (`write_q`) to hold tag data, as shown in Figure 3, without concern about the system call synchronization or the reflected memory location in the tag recipient. This is because the TCP channel is an ordered byte stream, from the point of view of a process.

We instrument `read()` and `write()` system calls and the following post operations are performed before the calls are returned[5].

- `write()`: push tag data into the `write_q` of a corresponding channel (fd) for the successfully transferred data of size w and its address range (`&w_buf[0]` – `&w_buf[w-1]`) in order.
- `read()`: for the successfully received data of size r, request corresponding tag data from the peer's flow manager. The flow manager pops up the requested data from the `write_q` and sends to the recipient. Then, the recipient reflects the tag data to the appropriate address range (`&r_buf[0]` – `&r_buf[r-1]`).

Since the TCP channel is ordered, we can expect that there are already pushed tag data of size w', accumulated and associated with multiple `write()` calls made previously in the sender, when a `read()` system call returns r and $w' >= r$.

3.1.3 Tag Resolution

As our framework is decentralized, the SeeC instance of each process maintains its own mappings between tag data and information source (a source information is mapped to a "color" value of a tag). Therefore, it is necessary to perform color value resolution for the newly introduced tag data with the data sender. SeeC makes queries for this purpose to the peer flow manager.

3.1.4 Threat Analysis

It is worthwhile considering the security implications of an adversary targeting our architecture.

Evasion: As our architecture depends on the underlying DTA tool for information tracking, attackers may possibly evade the framework by somehow avoiding the taint propagation tracking. The discontinuation is possible if the DTA tool does not cover all the instructions involved in information flow. It is also possible that an attacker may exploit dependencies other than those supported by the DTA tool. As discussed earlier, SeeC does not currently track implicit information flow. Fortunately, malwares that exploit such implicit flow have not been reported to date. However, we plan to enhance traceability by adding support for control dependencies and to increase the coverage of machine instructions supported by SeeC.

Subversion: Our framework needs to protect communication between flow managers by adding mutual authentication and tag encryption facilities, to prevent active or passive attacks by attackers in the network. Given that our architecture is intended for use within a single enterprise, the problem of key management is relatively straightforward; to protect the actual communications, a protocol such as TLS/SSL or IPsec is sufficient.

Members that are under an attacker's control cannot be trusted to provide truthful DTA information. Determining the trustworthiness of a system is an extremely hard problem, with no known solution to date. Our architecture does not currently address the issue of operation within a partially compromised/hostile environment.

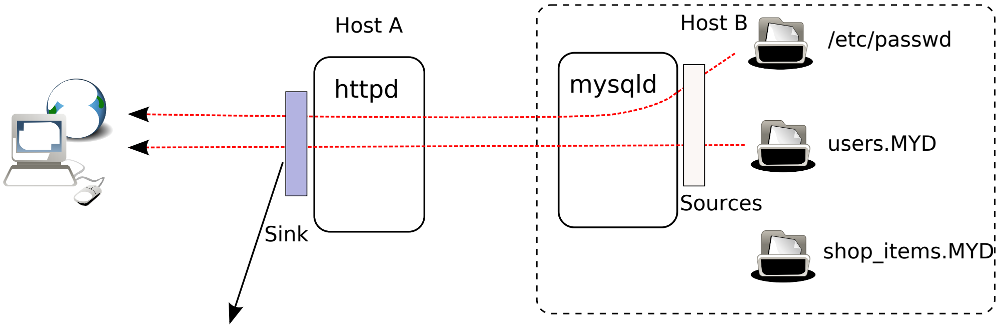
4 Experiments

In this section, we describe an experiment demonstrating distributed information flow tracking, wherein we have captured information leakage by an SQL injection attack involving web and database servers residing in different hosts.

4.1 Identifying Information Leakage by a SQL Injection Attack

Our experiment used the Apache web server (version 2.2.9) with PHP script support (version 5.2.6). For database server, we used MySQL (version 5.0.51). We attached SeeC to the Apache (`httpd`) and MySQL server daemons (`mysqld`) for information flow tracking, using the concatenated DTA facility described earlier.

[5] The same actions apply to `readv()` and `writev()`.



```
@@@ Tainted Memory Area [0x8296510 + 7266]
Source Info ---
COLOR: 0x2 [00000000000000000000000000000010]
128.59.19.220:33963
COLOR: 0xc [00000000000000000000000000001100]
128.59.19.225:[/home/chan/eggbox/mysql/mysql-5.0.51b/var/test_db/users.MYD]
128.59.19.225:[/etc/passwd]
Sink Info ---
FD(11) # NETWORK SOCKET
Local [IP: 127.0.0.1, PORT: 8080, LISTEN:0]
Peer [IP: 128.59.19.220, PORT: 33963]
@@@ ----
```

Figure 4. Information leakage by a SQL injection attack.

We set up an example SQL injection attack, shown in Figure 4. The web server uses the information from the database table `users`, actually stored in the file `users.MYD` in the host machine of the database server, for authentication purposes, *e.g.,* matching the asserting username and password of a user trying to log in. Obviously, that information should not be leaked outside of the web server boundary. Once a user logs in successfully, a web page displays shopping items for customers through a PHP script. Shopping items are stored in the `shop_items` table (`shop_items.MYD`), and the contents are retrieved with the following simple query string:

```
select * from shop_items where id='$id'
```

The page fails to check the user-supplied string from the login web form; thus, an attacker can successfully *inject* the following string via the web form associated with the PHP variable `$id` to extract user information from the database:

```
' union select 1,name,passwd,1,
load_file("/etc/passwd") from users #'
```

Therefore, the processing results in the following (exploited) query string that includes the attacker-supplied additional *select* phrase:

```
select * from shop_items where id=''
union select 1,name,passwd,1,
load_file("/etc/passwd") from users #''
```

Although the original service would list shopping items from the `shop_items` table, the exploited query string retrieves name and password data from the `users` table together with the contents of the `/etc/passwd` file of the system on which the database server runs.

To detect such an information leakage, we set up taint sources and sinks as follows:

Taint sources: We have specified taint sources on the database server host to be any database (*.MYD) and the `/etc/passwd` file. The MySQL server reads such information with `read()` or `pread()` system calls.

Taint sinks: Any network output streams in the web server are sink points. On the web server, HTTP replies are sent via a `writev()` system call.

With a test attack request, we could successfully identify the original information sources at the sink point: Figure 4 includes a log of such sink capturing in the `writev()` system call (one of `iovec` structures). With the colored tainting, sources in the database server host are discriminated in the sink point of the web server.

During the experiment with the MySQL server, we also identified a *taint laundry* effect. As our test query involves a string function `load_file()`, the query processing first reads the `users` table and the `/etc/passwd` file, then prepares the response by writting its contents to a temporary file. Therefore, the taint propagation is discontinued because the result, forwarded to the web server, is from the temporary file which is not associated with a taint label that concerns us. We built a makeshift connection for this experiment to avoid the taint laundry effect. In practice, we

Table 1. MySQL sql_bench performance: The unit for the create_MANY_table benchmark is table (*i.e.,* tables created). The unit for the other benchmarks is record row.

Benchmark	Quantity	Native	With SeeC
create_MANY_tables	10000	15s	43s
insert	30000	29s	604s
select_cache	10000	57s	3447s

would be using filesystem features such as Extended Attributes to keep track of tainted information that has been written to the filesystem.

4.2 Performance

To evaluate the overhead introduced by SeeC, we have tested it with a heavily CPU-oriented application: gzip compressing 261MB of Linux kernel source code in a lightly loaded machine. We have two implementations that differ in the shadow memory structure. The incurred overhead with the simple, naive approach was approximately 28.1X, while with the page-table-like structure it was about 15.2X. For reference, a simple PIN-based tool that only instruments instructions and does nothing for analysis incurred 1.28X. This shows that there is significant scope for improvement in our tool.

MySQL is a complex multi-threaded server application. We ran a benchmark test with sql_bench tool for a MySQL server supervised by SeeC. Table 1 shows results from benchmarks related to creating tables, inserting and selecting records. Our test runs show 2.9X, 25.2X, and 60.5X overheads for each test case respectively.

To see the penalty of instrumenting a TCP channel for concatenated tainting process, we transferred a 1MB size random file over netcat (nc) and measured the completion time. The overhead was approximately 190X, which is not surprising given the need for multiple system calls for each actual data transfer.

In the current unoptimized framework, the overhead comes mostly from instrumenting the TCP channel: hooking I/O-related system calls in the application layer and transferring taint tag information to the connecting peers. The transfers also involve interaction with other processes (flow managers). We could enhance the performance by normalizing tag information and only sending limited-size representative tags, and using an in-kernel infrastructure for delivering addtional tag information. For the performance of DTA, some works focus on the issue [13, 19].

5 Related Work

DTA Implementations Recently, there has been much work on implementing DTA with DBI frameworks or whole-system emulators. TaintCheck [12], LIFT [13], Dytan [14] and Flayer [15] are implemented using DBI frameworks such as Valgrind [3], StarDBT [4], and PIN [2]. In contrast, TaintBochs [17], Argos [16], and Panorama [18] are implemented on whole-system emulators such as Bochs [6] or QEMU [7]. Because software-based implementations typically incur significant performance overhead, there have been efforts to implement DTA in hardware [8, 11, 9, 10].

DTA Uses TaintCheck and Argos are designed to detect previously unknown control-hijacking attacks, such as buffer overflow or format string vulnerabilities, and also include signature generation features. Sweeper [21] and Vigilante [22] used DTA to analyze worm viruses and generate antibodies or alert information. TaintBochs and Panorama showed how DTA can be used to capture information flow of specific real-life applications in system-wide view. Egele *et al.* [20] applied DTA to monitoring behavior of BHO objects in Windows, to detect privacy-leaking spyware.

There has been work on adopting static or dynamic taint propagation (analysis) as type systems to perceive information flow for confidentiality or integrity purposes. Shankar *et al.* [27] applied static taint analysis, extending the C language type qualifier, to taint or untaint program variables to detect format string attacks. Huang *et al.* [24] extended the PHP language to track information flow by inserting type qualifier. Data variables, associated with types, are classified with a lattice structure thereby preventing integrity or confidentiality violations in web applications.

SQL Injection and DTA As SQL injection attacks have taken off in recent years, there have been efforts to secure web applications against SQL injection using information flow techniques. Lam *et al.* [25] applied sound static information flow analysis and model checking to detect taint-based vulnerabilities. Halfond *et al.* [23] proposed *positive tainting*, *i.e.,* tracking trusted data, as opposed to normal *negative tainting*, *i.e.,* tracking unstrusted data. The propagated trusted data flow is evaluated (before it is used as an SQL string) in a syntax-aware manner to reduce false positives. Nguyen-Tuong *et al.* [26] modified the PHP interpreter to include a precise taint propagation facility. Against SQL injection, they check whether operators, keywords, or identifiers in SQL strings are tainted.

6 Conclusion

We have described SeeC, an experimental proof-of-concept framework for tracking information flow across system boundaries. SeeC tracks information flow within a single process by propagating taint during instruction execution, and uses a flow manager to arbitrate tag delivery by concatenating the data sources and sinks of communicating processes. We have shown a simple scenario with an SQL injection attack that involves unintended information leakage.

Our current implementation is a proof of concept. We plan to design a new architecture with performance in mind, and to apply the decentralized information flow model to enforce confidentiality and integrity policies across a distributed enterprise.

Acknowledgments

We would like to thank James Clause, Seung Jin Lee, and Sambuddho Chakravarty for their helpful comments and assistance. This work was supported in part by a research gift from Intel Corporation, and by the National Science Foundation under Grant CNS-06-27473. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the US Government.

References

[1] IBM ISS X-Forece, “IBM Internet Security Systems X-Force 2008 Mid-year Trend Statistics,” Jul. 2008.

[2] C. 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,” Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 190–200, 2005.

[3] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” SIGPLAN Not., Vol. 42, No. 2, pp. 89–100, 2007.

[4] C. Wang, S. Hu, H. Kim, S.R. Nair, M.B. Jr., Z. Ying, and Y. Wu, “StarDBT: An efficient multi-platform dynamic binary translation system,” Proc. Asia-Pacific Computer Systems Architecture Conference, pp. 4–15, 2007.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP), pp. 164–177, 2003.

[6] http://bochs.sourceforge.net/

[7] F. Bellard, “QEMU, a fast and portable dynamic translator,” Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track, pp. 41–46, 2005.

[8] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” Proc. of the 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 85–96, 2004.

[9] J. Kong, C.C. Zou, and H. Zhou, “Improving software security via runtime instruction level taint checking,” Proc. of the 1st Workshop on Architectural and System Support for Improving Software Dependability, pp. 18–24, 2006.

[10] M. Dalton, H. Kannan, and C. Kozyrakis, “Raksha: a flexible information flow architecture for software security,” Proc. of the 34th International Symposium on Computer Architecture, 2007.

[11] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Flexitaint: a programmable accelerator for dynamic taint propagation,” Proc. of the 14th International Symposium on High-Performance Computer Architecture (HPCA), pp. 173–184, 2008.

[12] J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” Proc. of the 12th Symposium on Network and Distributed System Security (NDSS), 2005.

[13] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou and Y. Wu, “LIFT: A low-overhead practical information flow tracking system for detecting security attacks,” Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 135–148, 2006.

[14] J. Clause, W. Li and A. Orso, “Dytan: a generic dynamic taint analysis framework,” Proc. of the International Symposium on Software Testing and Analysis, pp. 196–206, 2007.

[15] W. Drewry and T. Ormandy, “Flayer: exposing application internals,” Proc. of the 1st USENIX Workshop on Offensive Technologies (WOOT), 2007.

[16] G. Portokalidis, A. Slowinska and H. Bos, “Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation,” Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), pp. 15–27, 2006.

[17] J. Chow, B. Pfaff, T. Garfinkel. K. Christopher, and M. Rosenblum, “Understanding data lifetime via whole system simulation,” Proc. of the 13th USENIX Security Symposium, 2004.

[18] H. Yin, D. Song, M. Egele, C. Kruegel and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” Proc. of the 14th ACM Conf. on Computer and Communications Security (CCS), pp. 116–127, 2007.

[19] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, “Practical taint-based protection using demand emulation,” Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 29–41, 2006.

[20] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, “Dynamic spyware analysis,” Proc. of the USENIX Annual Technical Conf., No. 18, 2007.

[21] J. Tucek, S. Lu, C. Huang, S. Xanthos, Y. Zhou, J. Newsome, D. Brumley, and D. Song, “Sweeper: a lightweight end-to-end system for defending aginst fast worms,” Proc. of the 2st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), pp. 115–128, 2007.

[22] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang and P. Barham, “Vigilante: end-to-end containment of internet worms,” Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP), 2005.

[23] W. Halfond, A. Orso, and P. Manolios, “Using positive tainting and syntax-aware evaluation to counter SQL injection attacks,” Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 175–185, 2006.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” Proc. of the 13th international conference on World Wide Web (WWW), pp. 40–51, 2004.

[25] M.S. Lam, M. Martin, B. Livshits, and J. Whaley, “Securing web applications with static and dynamic information flow tracking,” Proc. of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PERM), pp. 3–12, 2008.

[26] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley. and D. Evans “Automatically hardening web applications using precise tainting,” Proc. of the 20th IFIP International Information Security Conference (SEC), pp. 295–308, 2005.

[27] U. Shankar, K. Talwar, J.S. Foster and D. Wagner, “Detecting format string vulnerabilities with type qualifiers,” Proc. of the 10th conference on USENIX Security Symposium, 2001.

[28] W. Chang, and C. Lin, “Guarding programs against attacks with dynamic data flow analysis,” Proc. of the 7th Annual Austin CAS International Conference, 2005.

[29] S. McCamant and M.D. Ernst, “Quantitative Information-Flow Tracking for C and Related Languages,” Computer Science and Artificial Intelligence Laboratory Technical Report, MIT, MIT-CSAIL-TR-2006-076, 2006.

[30] J. Seward and N. Nethercote, “Using Valgrind to detect undefined value errors with bit-precision,” Proc. of the 2005 USENIX Annual Technical Conference, pp. 17–30, 2005.