Towards Practical Avoidance of Information Leakage in Enterprise Networks

Jason Croft Matthew Caesar
University of Illinois at Urbana-Champaign
{croft1, caesar}@illinois.edu

Abstract

Preventing exfiltration of sensitive data is a central challenge facing many modern networking environments. In this paper, we propose a network-wide method of confining and controlling the flow of sensitive data within a network. Our approach is based on *black-box differencing* – we run two logical copies of the network, one with private data scrubbed, and compare outputs of the two to determine if and when private data is being leaked. To ensure outputs of the two copies match, we build upon recent advances that enable computing systems to execute deterministically at scale and with low overheads. We believe our approach could be a useful building block towards building general-purpose schemes that leverage black-box differencing to mitigate leakage of private data.

1 Introduction

Many modern networks contain highly sensitive and private data. Health care companies must securely store and protect private health records, and enterprise networks contain large amounts of corporate intellectual property that should not be leaked outside their network. Software on these networks may suffer from vulnerabilities that an attacker can exploit to *exfiltrate*, or leak, sensitive data. In protecting against such malicious or accidental leakages of private data, the ability to confine data within a network can be a powerful tool. However, confining data to a single machine on a network is not sufficient for the design of many corporate networks. These networks may contain file servers or content management systems that require data to be accessible inside—but not susceptible to leakage outside—the network. A solution that only protects a single node may also severely limit the usability of applications that depend on the sharing of private data.

In addition to the threat of external attacks, companies must also train and trust users to understand and obey policies regarding these types of sensitive data. A recent ISACA survey [1] revealed 35% of corporate employees have *knowingly* violated corporate information flow policies at least once, and 22% have transferred sensitive internal information using external storage devices. While operators of these networks invest in high levels of security to prevent intrusions (unauthorized access that can alter internal network state) and extrusions (unauthorized leakage of private information externally), several recent high-profile events indicate that we need new approaches to deal with these problems [8, 21, 25].

While previous work has explored the problem of information leakage, most recent research [17, 20, 26, 28] has focused only on protecting a single-node—where data is completely confined to one machine—and not the larger problem of network extrusion. The latter problem is more complex; here, private data can be transmitted within the network, but must be prevented from leaving the network. Hence, simply tagging and tracking private data within a single machine is insufficient. In addition, these systems typically rely on information flow and taint tracking [16, 17, 18, 26], which incur significant performance and memory overhead (a factor of 20 slowdown or more in some cases). While there are commercial solutions [2, 4, 5] that perform deep packet inspection, these systems may not be able to monitor encrypted traffic without encryption keys or information flows that are intentionally obfuscated by attackers.

An alternate promising direction is the use of *black-box differencing* to detect information leakage [12, 20, 27], in which a process's dependence on private data is determined by duplicating the process and supplying different input to the copy. In order for the two copies to be comparable in the absence of private data leakage, their runtime behavior needs to match. To achieve this, these works require *deterministic execution*, a collection of techniques (*e.g.*, a software layer) which eliminate randomness (nondeterministic behavior) during execution of the process. Solutions based on black-box differencing have lower overhead than information flow and

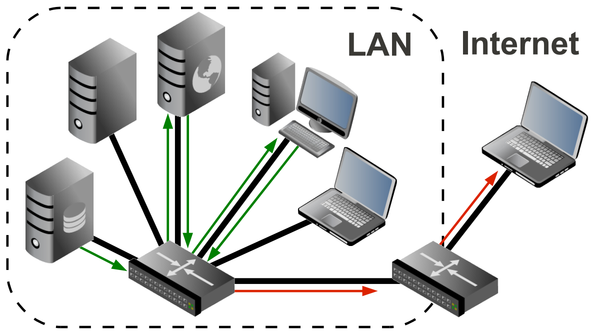

Figure 1: An enterprise network (denoted by dashed lines) containing various servers storing sensitive information and client machines requesting data. Green arrows depict flows of sensitive data flowing within the network to trusted machines, and red arrows show leaks of this data outside the network.

taint tracking techniques [16, 17, 18, 26] by more than an order of magnitude. However, existing solutions focus on single-machine scenarios—and hence are not directly applicable to network-wide confinement—and assume processes execute determinstically.

To address this, we propose a *network-wide* method of confining and controlling the flow of sensitive data, by extending black-box differencing techniques. While such an approach seems challenging, given recent breakthroughs in performing deterministic execution with low overheads and in distributed settings [10, 14, 22], we believe this approach is worth revisiting. Instead of copying (and scrubbing data from) a single machine, our approach creates a logical *shadow* copy of the entire network under consideration, and private data is scrubbed from the shadow copy. In practice, this is done by having each machine in the network create and maintain shadow processes in a coordinated fashion. Output from both the original and shadow processes is sent onto the network. If this data is destined for a machine on the network (*i.e.*, a trusted node), the output from both processes is propagated to the original and shadow processes at the next-hop machine. Output from the processes is “paired” and sent within the same grouping of network packets, then parsed by the receiving machine. If the data is destined for an address outside the network, output is compared for divergence. When output diverges, only the output from the shadow process is sent outside the network.

Limitations: We make several assumptions in our design. First, we assume all machines within the enterprise network are trusted. That is, we do not attempt to protect a machine from being accessed by another local machine that should not have access, but assume another form of access control is in place to protect sensitive data from clients within the network. For example, consider a version control repository. If a user has checked out a copy of the code, our design will protect against malware attempting to leak data from the user’s machine or from the server hosting the repository. It will not protect against a user on the network trying to gain access to the code who should not have access, but would protect from this user attempting to exfiltrate this data outside the network once she has access. Second, we assume all communicating machines within the network support our modifications to the Linux kernel. Data from original and shadow processes are grouped together, and a machine running a standard kernel would not understand how to parse such a packet. We are currently investigating methods for our design to be more incrementally deployable and support machines that may not run our modifications.

2 Related Work

Most recent research in information leakage has focused only on confining sensitive data to a single node within a network. This prohibits the sharing of sensitive data within an enterprise network and adversely affects many applications that require such sharing of data. From Tightlip [27] we borrow the idea of forking a copy of a process reading private data and *scrubbing* the data to remove the sensitive input to the copy. Output from the two processes is examined to detect dependence on the private data and signal warnings of potential leaks. However, Tightlip’s design only confines data within a single machine—upon detecting divergent output the kernel either blocks the network output or allows the non-sensitive copy to send data. Capizzi et al. confine data in a similar manner, but remove sensitive input and compare output from two copies of the same VM rather than individual processes [12]. Privacy Oracle [20] also employs a similar technique, which the authors term *differential black-box fuzz testing*, to monitor perturbations in network traffic from different inputs. However, the system requires executing the application and rolling back to re-execute with different input—providing no real-time protection—and the algorithm to detect divergent output in network is highly sensitive to reordering of packets.

Another recently common approach to mitigating information leakage is information flow tracking [15, 23]. However, static information flow techniques require access to source code and therefore cannot support legacy software. To address this limitation, dynamic taint analysis has been employed to track the propagation of private information throughout a system, at the cost of high overhead [13, 17, 16]. However, techniques such as those in [16, 17, 26, 28], cannot allow sharing of private data within a confined network, and incur slowdowns of 20X or more. Ermolinskiy et al.’s practical taint-tracking (PTT) [18] improves on some prior taint tracking approaches, but nevertheless has a slowdown of a factor of 22.

Commercial solutions to "data loss prevention", such as those by RSA [4], McAfee [2], and Symantec [5], rely on deep packet inspection to match regular expressions or keywords for known sensitive data (*e.g.*, credit card or social security numbers), and may not be able to monitor encrypted traffic without encryption keys or obfuscated information flows. Web content filters such as Websense [6] limit where hosts can send data, but can be circumvented by attackers by using public sites (*e.g.*, Wikipedia to post and retrieve sensitive data. Borders improve on some of these limitations by computing the expected content of HTTP requests using external information, including previous network requests, server responses, and protocol specifications, to measure leakage capacity [11].

In addition, Mandatory Access Control (MAC) can be used to control flow of sensitive information using policies to prevent subjects with access to sensitive data from communicating over a network. However, these policies can be difficult to set and can limit the functionality of many applications [19]. Borders et al. use *storage capsules* to provide a similar level of security as MAC using snapshots taken before sensitive, encrypted data is accessed, and reverting all changes within the system except those within the data capsule. However, switching between two states–one where secure data is accessed and edited, and one where it is not—can have lengthy transitions of up to 20s.

3 Motivation

3.1 Lack of Network-Wide Protection

In previous (single-machine) approaches to black-box differencing, data cannot be easily shared within the network. This can severely limit the usability of applications and prevent sharing of private data where sharing might be necessary (*e.g.*, a source code repository). A network-wide solution to confining data leaks requires more control of the flow of information with respect to the data's destination. Output from both original and shadow processes must be preserved across machines to prevent additional leaks of data.

For example, consider a process P_2 executing on a machine M_B that outputs the size of a string in bits, as shown in Figure 2. Suppose in each case a simple scrubbing mechanism is used that replaces private data with random data of the same length, and P_2 reads a file whose size is considered sensitive data. Another process P_1 on machine M_A reads a private file containing the bits `11`, applies some transformation, and sends the data to P_2. M_A's kernel will spawn a shadow process and pass some scrubbed input to this shadow process, such as `00`. Suppose the output of P_1 is `110` for the original process and `00` for the shadow process. If P_2 were to receive only the output of P_1's original process and then scrub the input itself, transforming `110` to `000`, the output from P_1 and its shadow process will both be 3. However, if we envision the processes across the two machines as a single process on a single machine, the output of P_2's shadow process should be 2. Thus, if the output of the shadow process is discarded and the output of the original is naively re-scrubbed at the receiver, data could be leaked. An attacker knowing this limitation could easily contrive a malicious program to leak these small amounts of data and perhaps reconstruct part of the original sensitive file.

3.2 Deterministic Execution

Our work is also motivated by recent advances in deterministic execution [7, 9, 10, 14]. Though not necessarily required for our design to be effective, leveraging techniques providing deterministic execution can reduce false positives in detecting divergent output. This ensures differences in output to the same program are solely the result of different inputs.

dOS [10] introduces a new OS abstraction termed *deterministic process groups* (DPG) and ensures all communication between processes in these groups are deterministic. A shim layer interposes on communication that crosses the boundary between the DPG and the rest of the system. DPGs are implemented within the Linux kernel, therefore integrating these abstractions with our implementation may be possible. Other designs providing system enforced deterministic execution include Determinator [7] and TERN [14].

The overheads of these three systems vary between 1.1X and 10X, which we believe is not yet fast and efficient enough in the worse case for deployment in enterprise networks. However, in the near future as these overheads continue to decrease, we can leverage these techniques in our system. By applying these mechanisms for system-enforced determinism, we can reduce the number of falsely detected information leaks from nondeterminisitic execution.

4 Design

4.1 Shadow Processes

Our design adopts a black-box approach to detecting output from a process that may contain sensitive data. It performs this detection by removing the sensitive output to a copy of the process and comparing output to the original. We cannot simply check output for the original sensitive data, as the process may encrypt, transform, or obfuscate the data. Therefore, upon reading sensitive data from disk or over the network, the process is forked and the input is changed. The *original process* receives the sensitive input while the copy, or *shadow process*, receives

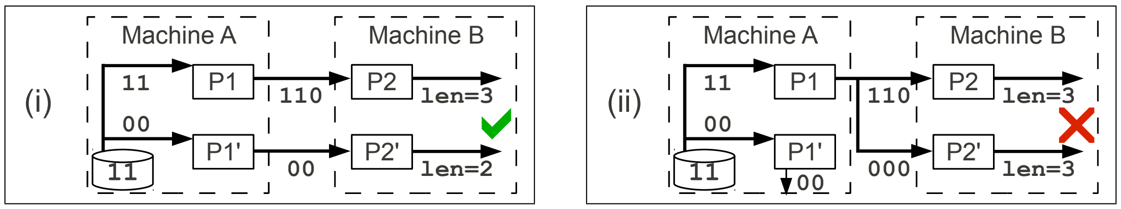

Figure 2: An example illustrating the need for shadow processes to send data within the network, as opposed to re-scrubbing input at the receiver. On Machine A, process P_1 applies a transformation to the private file containing the bits `11`, and on Machine B, process P_2 outputs the length of the output in bits. In (i) the shadow process P_2' sends its output the shadow process P_1' on B, and the outputs across P_2 and P_2' diverge. In (ii), the output of P_1' is discarded, and the output from P_1 received at B is re-scrubbed. P_2 and P_2' do not diverge, causing a false negative where data is leaked.

input where the sensitive data is removed, or *scrubbed*. Both processes then execute semi-independently of each other—in certain cases, the two processes must align to ensure any divergence in output results only from different input. For example, if both processes execute a system call to retrieve the time of day, the same result should be returned to both. To limit unintended sources of divergence (*e.g.*, a call to `gettimeofday`) or obtaining a seed for a sequence of pseudo-random numbers, we utilize a similar technique to Tightlip, in which the resulting value from one processes is shared with the other to ensure both receive the same result from the system call.

In Tightlip, a shadow process is spawned and the kernel forces the original and shadow process to align at each system call. Should the control flow of the two processes diverge (*e.g.*, due to input) then Tightlip will either kill one of the processes or swap the shadow with the original process so that no information can be leaked. However, a malicious attacker knowing this limitation could craft a process to circumvent this to leak data based on whether or not the process finished or was killed. Instead, our design allows processes to diverge and forces alignment only at system calls that may cause unintended divergence.

4.2 Sensitive Data

Sensitive files stored on disk must be marked sensitive, so the kernel can determine if a shadow process must be spawned each time a process accesses a file. A file's sensitivity must be a boolean value—either it is sensitive or not. We implement a file's sensitivity as an inode flag, which must be set manually by the user—a process that can be automated via policies or scripts (*e.g.*, based on file extension).

Upon reading a sensitive file from disk, the kernel will spawn a shadow processes for the reading process. Prior to forking the processes, a buffer is set up to share between the two processes and the result of the read call is scrubbed and copied to the shared buffer. Our current implementation uses random scrubbing—that is, the sensitive data is replaced with random, junk data of the same size. We use data of the same size to prevent odd behavior or crashes in applications due to unexpected input size.

In future work, we will investigate more advanced methods of scrubbing sensitive data to limit false positives and possible corruption of applications receiving scrubbed input. One caveat to our current scrubbing method is that input to the shadow process may crash or corrupt applications that serialize and store data to disk. For example, if a file contains phone numbers, scrubbing the input to the shadow process may cause it to diverge or crash if the scrubbed data is not a valid phone number.

4.3 Detecting Divergence

Network output between the two processes is compared for divergence and checked against the destination. If divergent output is destined for an address outside the network, only the output from the shadow process is sent. As this process has no access to the sensitive data, no data is leaked outside the network, thus confining sensitive data within the network. If divergent output is destined for an address within the network, output from the two packets is *paired* together and tagged to denote the packet containing data from both an original and shadow process. The packet can then be parsed by the receiver and the respective data can be delivered to the original and shadow processes, ensuring the state of both processes is maintained across the network.

While pairing packets incurs overhead on each machine within a network, this design has the benefit of reducing the complexity of additional network support (*e.g.*, special software on border routers to compare flows) and overhead on border routers. If both processes send data onto a network independently of each other, a mechanism is needed to guarantee no data from an original process is sent outside the network. This could be implemented on border routers by comparing flows between original and shadow processes to detect divergent output, and blocking the original's output. However, in an enterprise network, border routers may need to support throughput of a gigabit per second or more. With such high rates, comparing flows may be infeasible and large queues would be needed to store packets in cases where the flows between an original and shadow process are misaligned, perhaps due to reordering of packets on the network or delays.

Instead, we opt to offload the computation to compare output on client machines where processing power is higher than on border routers and less sensitive to minor increases in processing time for packets. For example, consider an enterprise network containing a remotely-accessible web server and a file server than can only be accessed internally. Our design allows the web server and file server—both trusted nodes within the network—to freely share sensitive data. When the data reaches a machine where the next hop is outside the network, the output from the original process is blocked in place of output from the shadow process.

4.4 Pairing Packets

Previous approaches to black-box differencing do not target support for sharing data within a network. These systems do not differentiate between data sent inside or outside the enterprise network, and there is no method of maintaining the state of shadow processes over a network. As we have shown in §3, maintaining the state of a shadow processes is needed to prevent leaks of data.

Therefore, we introduce the concept of *pairing* network output from an original and shadow process for data destined within the local network. We assume pairs of communicating machines support such paired packets, but do not expect such compatibility on remote machines, so data destined outside the network is sent only by shadow processes without any pairing. Since shadow processes have no access to private data, as it has been removed from the process's input, no sensitive data can be leaked outside the network. Processes that have no shadow process (*i.e.*, ones that have not accessed private data) can freely communicate outside the network and execute as though no changes to the underlying OS have been made.

Packets must be marked as containing data from multiple processes so a receiving process knows to split the data and deliver to two separate processes. Only a single bit is required and these paired packets are never sent outside the network. Therefore, there is some flexibility as we need not be concerned about all Internet routers supporting a change or dropping packets due to some invalid or reserved field being set. As such, in our current implementation we use the high-order bit of the IP fragment offset field since it is the only unused bit in the IP header. Thus, the first bit of the control flags in the IP header could be set to 1 for a paired packet, while the other two remain unchanged to denote if a packet can or cannot be fragmented and if it contains more fragments [3].

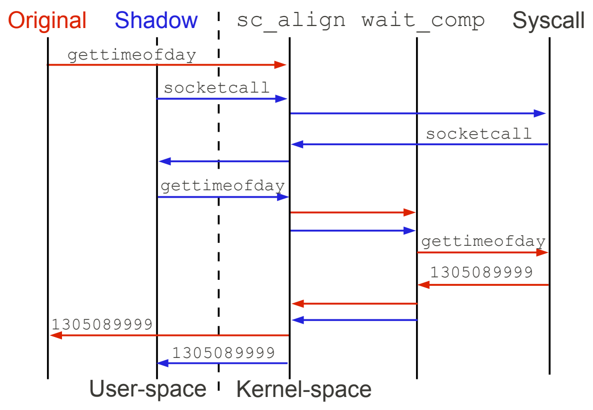

Figure 3: An original and shadow process execute and must align at a call to `gettimeofday` to prevent an unintended source of divergence. The shadow diverges in control flow, and the original waits until the shadow reaches the same system call. The original executes the system call and shares the result with the shadow, ensuring both receive the same result.

Figure 4 illustrates the pairing of output from two processes with output of the same length. In this scenario, equal sized chunks of data from each process are combined into a single IP packet, as well as the header for each process. The first 16 bits after the IP header, before the original packet's header, are reserved for the offset of the start of the second process's header. Should the size of the output from the two processes differ, and data from one process cannot be paired with output from another process, the packet is marked as a paired packet but with only the output from the one process inserted into the packet. The offset is set to 0 if there is no output from the original, or to the size of the original's header and data if there is no output from the shadow.

Preserving the output of shadow processes adds overhead in the number of packets that must be sent within the local network. For a process that is not sending or leaking any data outside the network (*i.e.,* the original and shadow processes do not diverge), the overhead is bounded at a factor of 2. If the shadow process diverges, and the divergent execution results in additional output compared to the original, overhead could exceed 2. This additional traffic should not significantly affect enterprise networks (data centers are a separate situation) as Pang et al. [24] have shown enterprise network traffic to be 2-3 orders of magnitude less than the capacity of the network. Therefore, doubling packet size or the number of packets that need to be sent should not have a significant effect on local network congestion.

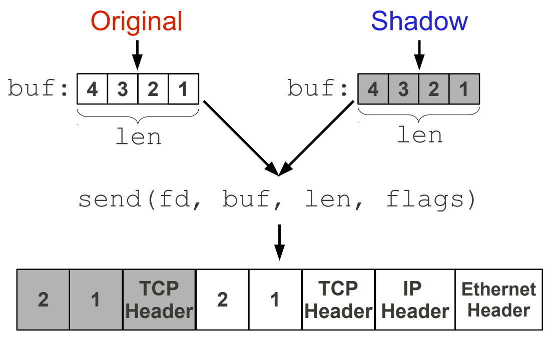

Figure 4: An original and shadow process execute a `send` system call. The data in their respective send buffers are placed in a TCP packet, and then both are combined within a single IP packet.

5 Conclusion

Threats of leaks of sensitive data are a growing threat to networks that store sensitive data, such as source code or customer information. To this end, this paper proposes a network-wide method of data confinement that detects information leaks by forking copies of processes consuming private data and removing the sensitive data from the input to the copy. We introduced the concept of a *paired packet* to allow both copies of the process to send data onto the network to allow the sharing of sensitive data within the confines of the network.

In future work, exploring more complex methods of scrubbing beyond simple replacement with random data would be interesting. Incorporating multiple levels of privacy (rather than the binary approach of private vs. non-private considered in this paper) may be worth exploring. In addition, though we do not believe our choice to implement our design as a modified Linux kernel would be an intrusive solution for network with high demands for security, a more system transparent design would be ideal. We plan to investigate methods of virtualization, beyond those discussed in [12], as possible starting points. Finally, we would like to investigate alternative approaches to deterministic execution to further reduce overheads of our design.

References

[1] ISACA Survey Reveals Alarming Computer Behavior. `http://www.isaca.org/About-ISACA/Press-room/News-Releases/2007/Pages/ISACA-Survey-Reveals-Alarming-Computer-Behavior.aspx`.

[2] Mcafee data loss prevention 9. `http://www.mcafee.com/us/enterprise/products/data_protection/data_loss_prevention/data_loss_prevention.html`.

[3] Rfc 791: Internet protocol - darpa internet protocol specification. `http://tools.ietf.org/html/rfc791`.

[4] Rsa data loss prevention suite. `http://www.rsa.com/node.aspx?id=3426`.

[5] Symantec data loss prevention. `http://www.symantec.com/business/products/family.jsp?familyid=data-loss-prevention`.

[6] Websense. `http://www.websense.com/`.

[7] A. Aviram, S. Weng, S. Hu, and B. Ford. Efficient system-enforced determinstic parallelism. In *OSDI*, 2010.

[8] J. Barnes. Pentagon computer networks attacked. *Los Angeles Times*, November 28 2008.

[9] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2011.

[10] T. Bergan, N. Hunt, L. Creze, and S. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[11] K. Borders and A. Prakash. Quantifying Information Leaks in Outbound Web Traffic. In *IEEE Security and Privacy*, 2009.

[12] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Preventing Information Leaks through Shadow Executions. In *ACSAC*, 2008.

[13] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.

[14] H. Cui, J. Wu, C. Tsai, and J. Yang. Stable Deterministic Multithreading through Schedule Memoization. In *OSDI*, 2010.

[15] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7), July 1997.

[16] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *USENIX ATC*, 2007.

[17] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[18] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. Mccauley. Towards Practical Taint Tracking. In *Technical Report No. UCB/EECS-2010-92*, 2010.

[19] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security*, 2003.

[20] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *CCS*, 2008.

[21] B. Knowlton. Military Computer Attack Confirmed. *The New York Times*, August 25 2010.

[22] C. Lin, M. Caesar, and K. van der Merwe. Towards Interactive Debugging for ISP Networks. In *HotNets-VIII*, 2009.

[23] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*, 1999.

[24] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A First Look at Modern Enterprise Traffic. In *IMC)*, 2005.

[25] N. Shachtman. Under Worm Assault, Military Bans Disks, USB Devices. *Wired*, November 19 2008.

[26] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *CCS*, 2007.

[27] A. Yumerefendi, B. Mickle, and L. Cox. TightLip: Keeping Applications from Spilling the Beans. In *NSDI*, 2007.

[28] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating System Review*, 45(1), January 2011.