Data Tethers: Preventing Information Leakage by Enforcing Environmental Data Access Policies

Charles Fleming, Peter Peterson, Erik Kline, and Peter Reiher
Computer Science Department
University of California, Los Angeles
Los Angeles, California 90095
Email: {fleming, pahp, icebeast, reiher}@cs.ucla.edu

***Abstract*—Protecting data from accidental loss or theft is crucial in today's world of mobile computing. Data Tethers provides flexible environmental policies, which can be attached to data, specifying security requirements that must be met before accessing that data. Data Tethers uses fine-grain data flow tracking to maintain these policies on derivative data. This is implemented by dynamic recompilation of legacy applications without the need to recompile from source. We demonstrate the system's feasibility with microbenchmarks that show individual component performance and benchmarks of real user applications like word processors and spreadsheets.**

I. Introduction

As computing devices become smaller and more mobile, data loss due to physical loss of a device becomes more and more of an issue for individuals, companies, institutions, and government agencies. Hundreds of thousands of sensitive records can be lost instantly when a laptop disappears from a coffee shop or a flash drive falls out of a bag.

Many organizations respond to this problem by mandating full-disk encryption for portable devices. While full disk encryption is useful in some cases, it does not help when a running laptop is stolen, or when the password that unlocks the encryption is weak. It also offers no mechanism to protect data that is sent over the network, or copied to non-encrypted storage devices.

In this paper, we describe Data Tethers (DT), a system that provides better control of information leakage by attaching environmental policies to data to specify conditions under which the data is safe from leakage. Data is stored encrypted, and is decrypted on access, only when the environment is deemed safe. When the environment is unsafe (e.g., when a laptop leaves a secure work environment), we remove sensitive data by encrypting it and destroying the key. Keys are stored remotely, and only given to the local machine when it can prove the environment is secure.

One issue with systems of this type is that it is difficult to insure that all copies of the data have the proper policy attached. Programs may make temporary copies or backups. Users may copy or cut and paste between files. Coarse grain solutions like attaching all policies associated with all data in use by the application result in policies attached to configuration or other unrelated files, over time rendering the system useless.

Dynamic information flow tracking (DIFT), a traditional solution for this problem, attaches a label to data and tracks all applications that touch the data, reattaching the policy when the data persisted. We use dynamic code rewriting techniques that work on arbitrary executables to implement a DIFT system to track labels at the word level. Thus, policies are applied only to copies or derivatives of data that should not be leaked.

II. System Design

A. Threat Model

The DT system is design to protect data in the event that a portable device is stolen by external attackers. We assume that the attacker has full access and control of the device, but cannot replicate or spoof the environment the device was designed to run in. Thus, it is not designed to protect against attackers internal to the organization who could conceivably operate the device in an acceptable environment. DT is built on top of existing access control, not as a replacement, and we rely on this to provide security for internal users.

We also assume that rightful users are non-malicious, and are not attempting to exfiltrate data or bypass the system. DIFT systems are, at best, always subject to covert channels, even such trivial ones as taking photos of the screen, and in the absence of special languages, cannot be made completely secure from leaks. The role of the DT DIFT system is to provide a mechanism to keep track of secure data in the system for clean up and correct labeling when persisting, and is designed to operate *only in a safe environment.* When the environment is unsafe, the policy enforcement mechanism should remove access to the data, and the DIFT system will never be used.

For similar reasons, we do not consider the case where the attacker steals the device, modifies it, and returns it to the owner. While we do verify that the operating system has not been tampered with on boot, regular user applications may be rewritten to leak information once the machine is back inside the safe operating environment. In this situation, the user should clean and reinstall the software on potentially compromised machines.

Another assumption we make is that the policy attached to the data is sufficient to detect that the current environment is safe in a timely manner. While secure data is always written to disk encrypted, it can be unencrypted in memory when the environment is safe. While DT cleans up secure data extremely quickly when an unsafe environment is detected, the transition from safe to unsafe cannot be detected instantaneously. This

transition interval is a window of attack, particularly for physical attacks such as cold boot. Proper selection of policy to minimize this window is essential.

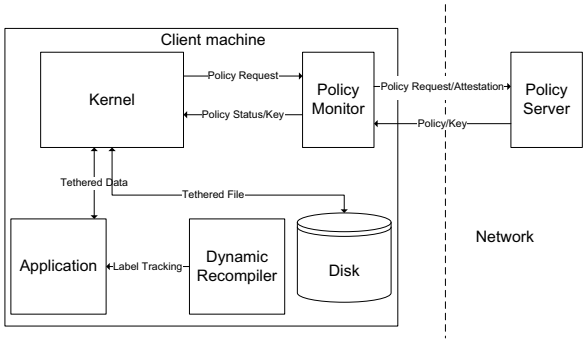

Fig. 1. Data Tethers high-level architecture.

B. Overview

The DT system is composed of three major components which can be seen in Figure **??**: the *policy server*, the *policy monitor*, and the *dynamic recompiler*. The policy server is an application that runs in a secure location and stores policies and keys for the data. The policy monitor runs on the local machine, monitoring environmental conditions, and handling communication with the policy server. The dynamic recompiler also runs locally and instruments applications that use tethered data. OS modifications support efficient tracking of data and handle policy violations.

Sensitive data is labeled with tags. When a user application reads this data from the disk, network, etc., the OS detects that tethered data is being accessed. If the relevant policy is not currently being monitored, the policy monitor retrieves the policy from the policy server and verifies that the policy conditions are met. If so, the policy monitor retrieves the encryption key for the policy from the remote policy server, passes this key to the operating system, and adds the policy to the list of currently monitored policies. The operating system starts the dynamic recompiler for the application accessing the data. The dynamic recompiler adds code to the application to track the data labels. Labeled data written to a file by the application is encrypted with the proper encryption key and tagged with the policy ID. The policy monitor periodically reevaluates the policy at policy-specified intervals. When a violation occurs, it notifies the operating system, which suspends all processes using affected data, encrypts that data, and destroys the local encryption keys.

III. Implementation

A. Platform

Our target platform is primarily single-user machines, particularly laptops and other portable devices which periodically leave the secure office environment. DT limits encryption and special handling to tethered data only, minimizing its performance impact. Operating system files, shared libraries, executables, and other files containing non-user data generally do not have policies attached, though it is not precluded for special cases where this is desirable.

B. Attaching Policies to Data

Policies are attached to data in three ways, depending on the data's state. First, we prepend files with a unique 256-bit marker followed by one or more policy IDs for the data contained in the file. For network streams, policy-controlled segments begin with a unique 256-bit marker, followed by a start tag that includes one or more policy IDs, followed by data in encrypted form, and closed with an end tag. Third, data in user space memory is labeled at the word level, with one word of label per data word. Each bit of the label indicates the presence/absence of a particular policy, which limits the number of policies per process; but for most cases this is sufficient and is similar to previous work [**?**], [**?**], [**?**], [**?**], [**?**]. Labels are stored in the user process's address space, so no switch to privileged mode is required to propagate labels.

C. Policy Propagation

Policy labels must be propagated whenever the data is copied. The dominant previous approaches are specialized languages, specialized hardware, and dynamic code rewriting. While recent research has focused on specialized languages or hardware due to the perceived high cost of dynamic recompilation, a primary goal of DT was to demonstrate that this approach was practical in a real computing environment. Thus, we could not rely on special hardware or expect that every application be rewritten and proven correct, given the wide range of user applications available; nor was limiting the user to secure applications desirable.

With the dynamic rewriting approach, we attach to a running process and add instructions into the executable, augmenting load, store, and arithmetic operators with instructions to copy or combine labels. Rather than creating our own code to perform dynamic rewriting, we use the Dyninst library [**?**].

While running instrumented code can be costly, we validated two assumptions that make the approach feasible. First, our target platform is personal computers and user applications. Most of these applications use small percentage of a modern computer's processor time. Multiplying this small percentage by a factor of four, five, or even ten will not result in a noticeable degradation of the system. Second, for many of these applications, much of the code never touches protected data, but instead deals with rendering menus, buttons, animations, etc. By carefully identifying portions of the code that manipulate the data of interest, we can limit the impact of the instrumentation. There are few references for exactly how much processor time the average user application takes, or what percentage of instructions operate on actual user data, but we show in our performance results (Section **??**) that for several representative applications, our assumptions hold.

Policy data must be stored in memory, preferably in user space to avoid switching between user and kernel mode. We store labels at the word level, with one label word for every

word of controlled data. These labels are stored in "shadow pages" in user space memory. Shadow pages are only allocated for pages that actually contain labeled data.

Applications have two types of data flow. The first is explicit flow which includes loads, stores, arithmetic operations, etc., where data is directly combined or moved from one memory location to another. The second is implicit flow, where information is transferred by the control flow of the program. It is impossible to prevent certain types of information leakage via implicit data flow without the use of special languages that can guarantee noninterference; i.e., that we make no assignments to low security variables inside conditional clauses controlled by high security variables [**?**], [**?**]. However, for our threat model, information flow tracking never takes place in an unsecure environment.

D. Taint explosion

The issue of taint explosion has recently been a topic of discussion in papers such as [**?**] or [**?**]. While DT is not immune to taint explosion, it is largely unaffected by it. We are focused on short lived user applications such as word processors rather than longer running applications like web servers or databases. Also, since the operating system itself is aware of labeled data, operating system data structures do not become tainted, spreading it to other processes.

E. The Data Barrier Concept

Policy-controlled data in the DT system exists either in an encrypted, packaged form or unencrypted and labeled in process memory. Conceptually, we create a *data barrier* around a process, with any data crossing this barrier being converted from one form to the other. Because various devices use different interfaces in the Unix kernel, we implement the data barrier in different ways for different devices. For disks, we integrate at the VFS level, allowing us to be agnostic to the file system. For the network, we augment the SockFS interface, which is just above the transport layer, allowing us to send tethered traffic over any transport. Some types of peripherals are excluded from the barrier for practical reasons, such as the video or sound card.

F. Environmental Monitoring

DT specifies environmental conditions under which data is accessible. These may be security requirements such as the presence or absence of software such as virus scanners, some types of user identity verification, location, or almost any other measurable status. Due to the flexibility of policies, the policy monitor accepts pluggable modules, which are run in a sandbox, that can be downloaded when a particular policy element needs to be verified. Each policy element specifies the granularity of detection it requires, in terms of both the length of time between status checks and the length of time a violation may be tolerated. While many policy elements require strict enforcement, others may not, for usability reasons. The policy monitor is responsible for both initial verification that the system's state satisfies the policy and for notifying the system when environmental changes invalidate a policy.

G. Cryptography

Keys are stored permanently on the remote policy server, and are delivered to the policy monitor only on receipt of a signed certification of the system state, including an attestation that the operating system and DT components are uncompromised. Keys are stored only in kernel memory on the DT machine, never on disk, and are destroyed on notification of a policy violation or on shutdown, suspend, or crash. Data may be subject to multiple policies, in which case it is encrypted sequentially with all applicable policy keys, in numerical order by policy ID.

H. Policy Language

DT policies are written in an XML-based language. Each policy in our language begins with the policy ID, followed a list of modules and required parameters for those modules to allow access. Modules implement testable conditions, such as location, time of day, or user identity and may be combined in AND/OR relationships. The polling interval specifies how often the policy should be checked.

I. Policy Monitor

When a policy is first encountered by the operating system, it sends a verification request to the policy monitor for the policy ID. The policy monitor then requests the policy from the remote policy server, downloads any modules necessary to verify the policy, verifies that the current system state meets the policy's requirements and sends a signed attestation of this to the remote policy server. The policy server forwards the decryption key to the policy monitor, which hands the key to the kernel for data decryption. The policy monitor then begins monitoring the system state at the intervals prescribed in the policy. If the state becomes inconsistent with the policy, the monitor notifies the kernel which begins the cleanup process.

Several modules have been written to test various environmental conditions including time of day, elapsed time, and location. The most complex is the location monitor which uses a remote server to determine if the client machine is inside a local network.

J. Policy Violation

When a policy is violated, the kernel suspends all processes affected by the policy change and encrypts their affected memory. After it deletes the encryption keys for the violated policy from memory, it flushes any devices, such as the video or sound card, that have had labeled data written to them. We encrypt swapped pages containing labeled data with the appropriate keys, so once the keys are destroyed these pages are secure. Processes that use encrypted data are suspended to disk.

IV. Performance

We break the cost of running instrumented applications down into several costs: rewriting code, running instrumented code, file system and network changes, extra memory pages,

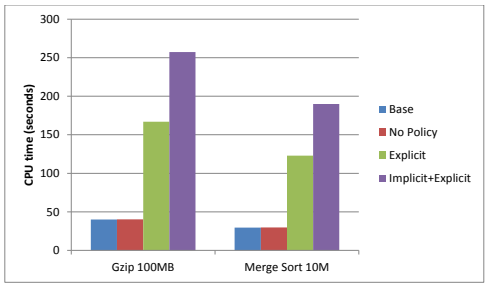

Fig. 2. Microapplication Benchmarks.

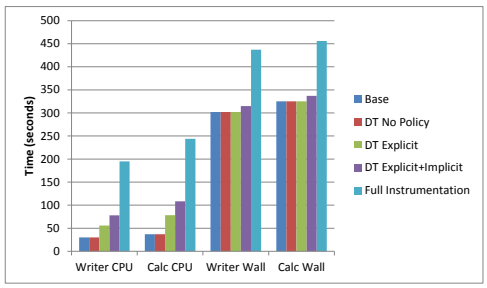

Fig. 3. Application benchmarks.

and watch points. We also evaluate the speed of cleanup following a policy violation. The machine used for our evaluation was a virtual machine hosted on a Sun T2000 server with an ULTRASparc T1 processor. Sixteen cores were allocated to the VM, with each core approximately as fast as a 1 Ghz Pentium 3 processor.

A. Correctness

While the correctness of an implementation is difficult to verify, we tested multiple cases, ranging from copying data from place to place in memory to combining data with multiple policies in various ways. All of the performance benchmarks discussed in the following sections were also checked for correctness. In all cases file labels were correct. We also verified that data with no policy attached was written unencrypted when the writing process had policy-controlled files open, and that data and keys were correctly cleaned up when policies were violated. We plan to investigate other methods of correctness verification.

B. Costs of Rewriting Code

The cost of rewriting code is low. The largest cost is Dyninst's initial parsing of the in-memory image of the application, which depends on the size of the application. The largest application tested, OpenOffice Writer, was around 100 megabytes in code size, including all libraries, and took less than 3 seconds total CPU time to instrument.

C. Costs of running instrumented code

To evaluate the cost of running instrumented code, we tested both microbenchmarks and user application benchmarks, measuring CPU time and total elapsed (wall) time for the base system, the DT system with non-policy controlled data, DT with explicit flow tracking only, DT tracking both implicit and explicit flows. For applications we also measured DT with full instrumentation of the executable. Our microbenchmarks are iterative versions of merge sort and gzip. These applications work almost exclusively with data and are instrumented in their entirety, representing the worst-case scenario for DT. We tested gzip with multiple file sizes, from a few kilobytes to several megabytes and merge sort with labeled input files from 1000 elements to 1 million elements, and averaged run times over multiple runs. These results are shown in Figure **??**. In both cases, the cost of instrumentation was approximately 4.1 times the base CPU time for explicit flow tracking and 6.4 times for tracking explicit and implicit flows. This was expected, considering the number of instructions added in the instrumentation process, increased cache misses and other side effects.

For application tests we chose two representative user applications: Writer and Calc, the OpenOffice word processor and spreadsheet. Typical use of these types of applications is hard to define, but we attempted to capture representative real user interactions using automated testing tools. These tools do not simply record and playback, but interact with the application in a similar way to a user, waiting for tasks to complete before continuing. We played this recording back, measuring CPU time and wall clock time, averaged over multiple runs, with the results seen in Figure **??**. All data in files used in the benchmarks was tethered.

The full instrumentation number shown in the figure is for the case where the complete application and all libraries are instrumented with both implicit and explicit flow tracking. This is similar to the case of the microapplication benchmarks, and indeed the performance penalty is similar, approximately 6.6 times the base, uninstrumented, CPU time. Elapsed wall time for this case is 1.45 times the base wall time. In contrast, when using selective instrumentation CPU time is only 2.6 times base for Writer and 2.9 times base for Calc, for the implicit and explicit flow tracking case. This indicates that with selective instrumentation, we instrumented only 28% and 36% of blocks that executed for Writer and Calc, respectively, a significant savings. Wall times for both applications were also only slightly affected by the selective instrumentation, requiring approximately 1.04 times the base wall time.

D. File System Benchmarks

For the file system, we performed a series of microbenchmarks to quantify the costs of our modifications to tethered and untethered files. Table **??** shows the results of our tests for both reading and writing both types of files. Care was taken to eliminate caching effects by using different files for each iteration of the test.

DT slows down the file system, primarily due to the need to scan memory for tags and to encrypt/decrypt. Approximately the same overhead is added to both reads and writes, but the base time for writes is much lower than for reads. Reads must wait for the disk read to complete before returning to the user,

TABLE I
FILE SYSTEM BENCHMARKS.

	Base	DT
Read 16k	14.6ms	31.0ms
Write 16k	2.7ms	18.0ms
Open	13.3ms	25.2ms

while writes may be completed asynchronously. Numbers for larger files are linearly related by size to the number shown for 16k files. Overhead is added for opens because an additional check is done on open to determine if the file is tethered. For most user applications, the overhead introduced is not noticeable compared to the runtime of the application, but applications that open and close many small files may see a performance decrease.

E. Network Benchmarks

For the network, we are mainly interested in the overhead of packing and unpacking tethered data, so our benchmarks measured time spent in the appropriate SockFS functions, not transmission time. Since network traffic always requires scanning for start tags, we benchmarked the DT system both for data with and without a policy attached. For these benchmarks we computed average time spent in the SockFS send and recv functions for small buffers of 256 bytes.

TABLE II
NETWORK BENCHMARKS.

	Base	DT/no policy	DT/policy
send	49 μs	69 μs	106 μs
recv	2127 μs	2204 μs	6623 μs

Table **??** shows that the cost of sending a packet of tethered data is approximately twice that for a nontethered packet, primarily due to scanning and encrypting the data. Receiving tethered packets costs slightly more than three times as much as an untethered packet. This difference between send and recv is due to buffering partially received decryption blocks. Very network-intensive applications may be negatively impacted by DT, but for most end-user applications, the overhead introduced is dwarfed by transmission times.

Also of interest is the data size overhead introduced when tethering streams. This varies depending on the degree to which the data is tethered. In the best case, where a stream is tethered with a single policy, the overhead is around 100-200 bytes. In the worst case, where every word is tethered with a different policy, or even multiple policies, this could balloon to 100-200 bytes per word. We anticipate that the worst case is very unlikely, and that most streams will contain few or no policies.

F. Paging and Page Protection Faults

Calculating the number of extra pages used by tethered applications is straightforward since one page is allocated for every page of tethered data. For applications like merge sort and gzip, there is one extra page per page of data, since all data is tethered. For the OpenOffice applications, it depends on the amount of internal data structures that are generated by opening the document. We found on average that for a 1 megabyte document where all data was tethered, around 3 megabytes of shadow memory physical pages were generated.

Also of interest is the number of page protection faults generated by the application. For gzip and merge sort, after the initial round of watch point faults, no additional page protection faults were generated since all code was instrumented at this point. For the OpenOffice applications, a small number of page protection faults were generated due to the application accessing non-watch pointed data on pages containing watch points, but the number was quite small, 0.4 faults per second on average.

G. Speed of Cleanup

To measure speed of cleanup we opened several applications with labeled data, and then forced a policy violation. Timing code was added to the kernel to measure the amount of time from the initial notification of the violation until the completion of cleanup. This took, on average, less than half a second.

V. RELATED WORK

Recently, there has been new interest in practical information tracking implementations in both hardware and software. Many hardware solutions have been proposed including RIFLE [**?**], Minos [**?**], Raksha [**?**], and Flexitaint [**?**]. These solutions generally involve special hardware built into processors to track tags in memory, similar in most cases to the labels used in DT but performed simultaneously with regular instructions. Many software information flow tracking systems have also been written, mostly focused on taint tracking to prevent buffer overflow type attacks. Some of the more prominent include Vigilante [**?**], Dytan [**?**], LIFT [**?**], and GIFT [**?**] as well as [**?**], [**?**].

Implicit flow tracking is an active area of research; however, much of this research is of limited use to us since it involves high level programming language abstractions and specialized language constructs. These systems include special security typed languages or language additions for augmenting code with information flow tracking, or specialized compilers. Sabelfeld and Myers [**?**] provide a good survey of this work.

FLASK [**?**] is a well-known operating system that provides flexible policy-based access control, similar to DT policies. Keypad[**?**] is a file system that tracks file access and allows for revocation of access when a device is stolen. Asbestos [**?**], Histar [**?**], and Flume [**?**] provide information flow control, labeling, and data isolation with high efficiency, at the cost of requiring applications written for those operating systems. [**?**] details a posture-based security system which does environmental monitoring similar to DT, but this system only limits what applications may be run under various conditions. The DStar system [**?**] extends the Histar model to distributed systems, allowing for transmission of labeled data between machines, however it is limited to tightly coupled machines

in a cluster environment. A data provenance system called GARM [**?**] uses static analysis to track provenance information. CLAMP [**?**] is a system with similar goals of preventing data leakage, but focuses on web applications using a LAMP-style development stack.

There have been two recent virtual machine based approaches that are similar in nature to DT, the S3 system[**?**] and the Neon system[**?**]. Both of these systems run the operating system inside of QEMU and convert native x86 assembly instructions into simulated instructions that do information flow tracking. While these are similar in flavor to DT, they suffer from several drawbacks in comparison. Since the information flow tracking happens at a higher level than the operating system, it is difficult to detect the types of environmental conditions we use in DT, for example, that a virus scanner is running or a specific user logged in. Also, since the operating system itself is instrumented it is possible for unrelated processes to become tainted, particularly since the operating system, by its very nature, is a long running process.

Another promising approach is that used in TaintDroid[**?**]. This system tracks the flow of sensitive user information inside Android applications by modifying the Dalvik JVM. While very low overhead was achieved with this approach, it is only applicable to applications that run inside a virtual machine, and not to native binary applications.

VI. Conclusion

DT offers a new method for organizations to prevent their valuable and sensitive information from being lost, by loss of portable machines and media. Unlike full disk encryption, DT can protect data even while a machine is running. While DT performance can be costly in the worst case, for many end-user applications the decrease in performance is not noticeable. DT works on legacy binary applications and does not require any changes in typical user behavior, except when such behavior would improperly leak data.

References

[1] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proc. of the 37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004.

[2] L. C. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," in *Proc. of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference (ACSAC)*, 2006.

[3] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting general security attacks," in *Proc. of Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[4] G. Suh, J. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2004.

[5] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[6] J. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic instrumentation for scalable performance tools," in *Scalable High Performance Computing Conference (SPHCC)*, 1994.

[7] M. L. Minsky, *Computation: finite and infinite machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1967.

[8] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *14th International Symposium on High Performance Computer Architecture (HPCA-14*, 2008.

[9] A. Slowinska and H. Bos, "Pointless tainting? evaluating the practicality of pointer tainting," in *Proc. of the 4th ACM European Conference on Computer Systems*, 2009.

[10] M. Dalton, H. Kannan, and C. Kozyrakis, "Tainting is not pointless," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 88–92, April 2010.

[11] ——, "Raksha: a flexible information flow architecture for software security," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 482–493.

[12] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005, pp. 133–147.

[13] J. Clause, W. Li, and R. Orso, "DYTAN: A generic dynamic taint analysis framework," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2007, pp. 196–206.

[14] B. Xin and X. Zhang, "Efficient online detection of dynamic control dependence," in *ISSTA '07: Proc. of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[15] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *Proc. of the Network and Distributed System Security Symposium (NDSS07*, 2007.

[16] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, 2003.

[17] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The FLASK security architecture: System support for diverse security policies," 1998.

[18] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, "Keypad: An auditing file system for theft-prone devices," in *Eurosys*, 2011.

[19] P. Efstathopoulos, M. Krohn, S. Vandebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazires, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos operating system," in *Proc. 20th ACM Symp. on Operating System Principles (SOSP*, 2005, pp. 17–30.

[20] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazires, "Making information flow explicit in HiStar," in *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 263–278.

[21] L. Zheng and A. C. Myers, "Dynamic security labels and noninterference," *International Journal of Information Security*, 2004.

[22] G. Durfee, D. K. Smetters, and D. Balfanz, "Posture-based data protection," *Technical Report, PARC*, 2007.

[23] N. Zeldovich, S. Boyd-Wickizer, and D. Mazires, "Securing distributed systems with information flow control," in *Proc. of the 5th USENIX Symposium on Network Systems Design and Implementation*. USENIX Association, 2008.

[24] B. Demsky, "Garm: Cross application data provenance and policy enforcement," in *ACM Transactions on Information Security*, 2011.

[25] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig, "CLAMP: Practical prevention of large-scale data leaks," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[26] S. Katti, A. Ermolinskiy, M. Casado, S. Shenker, and H. Balakrishnan, "S3: Securing sensitive stuff," in *USENIX OSDI 2008 Work in Progress Report*, 2008.

[27] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage, "Neon: System support for derived data management," in *Proc of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2010.

[28] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.