Dataflow Tomography: Information Flow Tracking For Understanding and Visualizing Full Systems

BITA MAZLOOM, University of California, Santa Barbara
SHASHIDHAR MYSORE, Eucalyptus Systems
MOHIT TIWARI, University of California, Berkeley
BANIT AGRAWAL, VMware
TIM SHERWOOD, University of California, Santa Barbara

It is not uncommon for modern systems to be composed of a variety of interacting services, running across multiple machines in such a way that most developers do not really understand the whole system. As abstraction is layered atop abstraction, developers gain the ability to compose systems of extraordinary complexity with relative ease. However, many software properties, especially those that cut across abstraction layers, become very difficult to understand in such compositions. The communication patterns involved, the privacy of critical data, and the provenance of information, can be difficult to find and understand, even with access to all of the source code. The goal of Dataflow Tomography is to use the inherent information flow of such systems to help visualize the interactions between complex and interwoven components across multiple layers of abstraction. In the same way that the injection of short-lived radioactive isotopes help doctors trace problems in the cardiovascular system, the use of "data tagging" can help developers slice through the extraneous layers of software and pin-point those portions of the system interacting with the data of interest. To demonstrate the feasibility of this approach we have developed a prototype system in which tags are tracked both through the machine and in between machines over the network, and from which novel visualizations of the whole system can be derived. We describe the system-level challenges in creating a working system tomography tool and we qualitatively evaluate our system by examining several example real world scenarios.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General

General Terms: Design, Management

Additional Key Words and Phrases: Dataflow tracking, tomography understanding

ACM Reference Format:
Mazloom, B., Mysore, S., Tiwari, M., Agrawal, B., and Sherwood, T. 2012. Dataflow tomography: Information flow tracking for understanding and visualizing full systems. ACM Trans. Archit. Code Optim. 9, 1, Article 3 (March 2012), 26 pages.
DOI = 10.1145/2133382.2133385 http://doi.acm.org/10.1145/2133382.2133385

1. INTRODUCTION

With each added layer of software abstraction developers gain more power to produce sophisticated systems, yet are further removed from the details of the underlying system. Few developers are cognizant of the many layers of software that run beneath and around their programs; the complexity of virtualization environments, operating

This work was funded in part by NSF Career Grant CNS-0910389, CCF-1005254.
Author's address: B. Mazloom; email: betamaz@cs.ucsb.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permission may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1544-3566/2012/03-ART3 $10.00
DOI 10.1145/2133382.2133385 http://doi.acm.org/10.1145/2133382.2133385

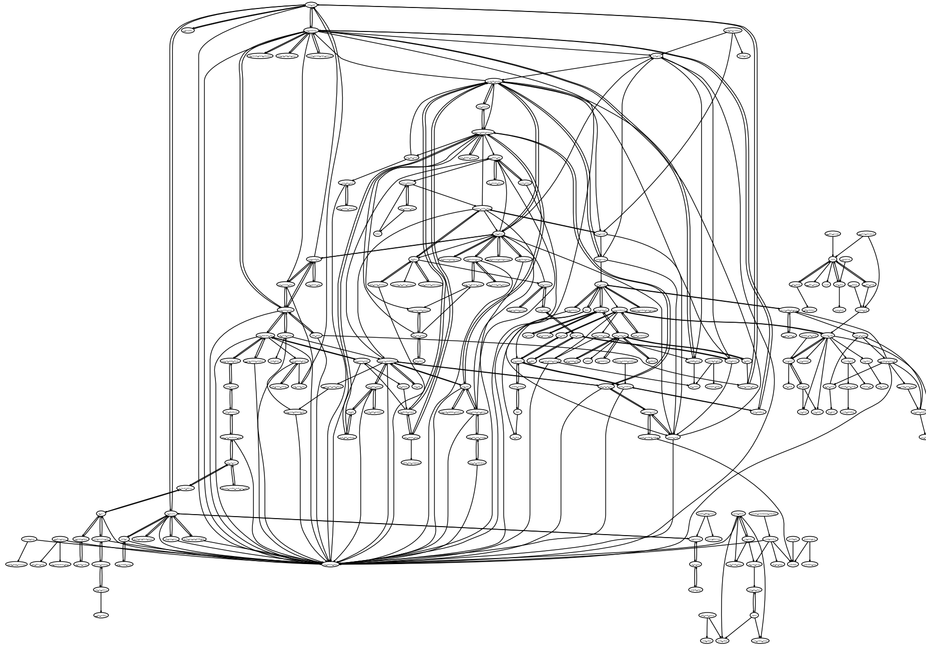

Fig. 1. The function-level control flow graph of a very simple server program that awaits a socket connection and replies “Hello World”. Even for such a simple program, there are 161 nodes (function invocations) and 409 unique edges (function to function call transfer). While control flow is an important part of the puzzle, it becomes increasingly difficult to understand interactions through control flow alone when multiple parties are trading information through sockets, kernel copies, and memory mapping. Data tracking allows the movement of particular bytes to be tracked between control boundaries allowing novel slices of the system to be generated.

systems, network services, third-party libraries, helper processes, and middleware, are all conveniently hidden behind a variety of interfaces. While this is necessary to the very idea of abstraction, the fact of the matter is that few people designing the system, and even fewer of those responsible for maintaining it, understand how all the pieces of the puzzle fit together.

While control graphs (call graphs, control flow graphs, and the like) are very useful, they are inherently tied to a single state: the program counter. With multiple services running at the same time, control flow can tell you about temporal ordering, but it is difficult to find true causation (e.g., Packet 1 caused Packet 2 to be sent over the network). The relationships between data-in and data-out are quickly lost, even in extremely simple network programs such as the one shown in Figure 1. By capturing a useful piece of the system dataflow, we can be certain to identify only true dependencies between events.

Our goal is to develop techniques that aid in the understanding of complex software systems, that go beyond static code visualizations, to shed light on exactly how a system is consuming, operating on, and propagating data throughout. We draw our inspiration from Positron Emission Tomography, in which a short-lived radioactive isotope is injected into a patient and the flow of the isotope is monitored through an ensemble of sensors. The resulting 3D image serves as a diagnostic map of the functional processes in the body. Rather than an isotope coursing through the veins of a human, we make use of various data information-flow tags running through the memory, registers, disks, and networks of a distributed system. By creating specialized

tag generation and propagation policies, customized to the unique needs of visualization rather than security, and tracking those tags at the ISA level, we can cut through many layers of abstraction yet provide detailed information directly related to the data in question.

Dataflow tracking for visualization has several advantages. By defining the tags at the level of the ISA, we need not assume any application will be directly assisting the process (e.g. through the use of a common request tagging middleware). Many systems are composed of software from a variety of vendors or projects, and an ISA level approach will still be able to trace data through these components, even if they were not written with such a tool in mind, because the semantics of the program are ultimately defined at the ISA level. This is also true for any number of compiled and interpreted languages. Furthermore, as long as tags are properly tracked through the application, the operating system, and over the network, many interesting communication patterns can be naturally identified. Regardless of whether data is transferred through shared memory, disk, OS copy, via UDP, or TCP sockets, identifying and visualizing communication simply becomes a matter of tag policy. Towards this end, we present the following contributions.

- We describe how tags can be generated by the network-interface, application, or instruction-rules, and tracked at the ISA level to aid the examination and visualization of systems composed of many independently developed components.
- We present three distinct classes of policy useful for dataflow tomography: a feed-forward tag-and-release model that finds data uses, a source-flow method that enables the system to track data back to its source, and a confluence method that relies on the collision of tags in the system to map boundaries between components.
- We implement a prototype of our system and describe how the ability to track tags between processes and over the network allows us to create policies that map the full procedure-level dataflow of a distributed system. We tested our method over a simple distributed system with a variety of components (OS, Mongrel, Ruby, Rails, Apache, Perl, and MySQL) and present visualizations of our results.
- We extend our initial prototype tagging infrastructure, where tags flow between physical memory and network device, to include tracking at the hard disk layer. Disk tracking provides the missing link needed for the many application that rely on the file system for temporary data storage, configuration, and even message passing.
- We provide a preliminary evaluation of the performance of our prototype by measuring the runtime of various popular applications such as compression, compilation and database accesses. For a clearer view of the effect tag tracking has on performance we compared different virtualization mechanisms to that of the native host.

While we describe our prototype system in a fair amount of detail, it should be noted, before we go further, that this paper is almost entirely qualitative. The ability to compose and reason about software of a significant scale is a serious and growing problem for all of computer science (during software development, software maintenance, education, etc.) yet it is very difficult to quantify. In this paper we argue that the an ISA-level view of the system gives us a uniquely advantageous position from which we can attack this problem, and rather than concentrate on architectural widgets that make such techniques faster or easier to implement, we have spent our effort attempting to elucidate the underlying ideas through examples (in Section 4), in a discussion of the numerous system level concerns (in Section 3), and in a simple performance characterization of our prototype system (in Section 5).

2. RELATED WORK

This work relies heavily on several lines of research including dynamic information-flow tracking, profiling, and distributed system tracing. In this section we describe the fundamentals of dynamic information-flow tracing, and we describe how we extend these works to allow for system tomography. We contrast our approach to other profiling and tracing techniques, which use time-based sampling, middleware level tagging, or other approaches to attempt to gain related insights.

Information-flow tracking has been a central idea to the security community for decades, yet since the introduction of the commercially unsuccessful Intel iAPX 432 in 1982, it has not seen support in commercial microprocessors. However, as transistor budgets continue to increase, many have argued that tagging is a natural, and minimally intrusive way of enforcing information-flow policies on modern CPUs. The basic idea behind a tagged architecture is that every piece of architecturally visible state, including the memory and register files, is extended with a corresponding bit or set of bits. Whenever an operation occurs, rules are used to determine how bits are propagated from source to destination, and policies can be enforced by restricting certain operations, depending on the value of these bits. For example, by treating all data that comes in from the network as "tainted," propagating the taint-bits through registers and memory, and preventing control flow from targeting tainted data, a broad class of code injection attacks can be prevented [Crandall and Chong 2004; Suh et al. 2004; Tiwari et al. 2009]. This technique can be naturally extended to propagate multiple independent bits [Dalton et al. 2007], to allow for general OS traps, to understand the lifetime of sensitive data [Chow et al. 2004], to detect and generate software exploits [Newsome and Song 2005], and to be used as an oracle [Crandall and Chong 2004] to analyze polymorphic worms [Crandall et al. 2005] and zero-day attacks [Portokalidis et al. 2006]. To alleviate the performance impact due to extra tag processing, tags can potentially be stored in a separate tag-cache [Venkataramani et al. 2007] with minimal changes to the memory hierarchy. Some have even proposed switching between the virtualized and emulated executions to improve performance [Ho et al. 2006]. There are also some completely software based schemes for information-flow tracking which employ source-level instrumentation [Xu et al. 2006] and binary code instrumentation [Newsome and Song 2005; Qin et al. 2006] to detect and prevent control flow hijacks and for worm containment [Costa et al. 2005]. Asbestos [Efstathopoulos et al. 2005] is a prototype operating system in which the application can convey a range of policies to enforce including permissible inter-process communication and the legal patterns for information-flow, and other researchers have considered the use of static analysis with runtime guards to enforce integrity and access control policies [Castro et al. 2006; Erlingsson et al. 2006; Zeldovich et al. 2006].

While dynamic information-flow tracking has a variety of uses for security and privacy, past techniques have concentrated primarily on its use as a mechanism by which policies can be enforced at runtime. Our work differs from this past work in both the underlying tag propagation rules that are used and in the overall intended use. Rather than using data tags as a mechanism for enforcement, we are instead using them as a means of discovering important behaviors and channels in the code. If general purpose hardware or low-level software support for data tracking is available to accelerate security policy enforcement, it could likely be used for our means as well. In addition, it is likely that our techniques may even be helpful in the design of information-flow tracking policies, especially in discovering and understanding exceptions to the standard rules.

Of course we are certainly not the first to propose techniques which aid software developers in understanding their systems in the presence of many layers of abstraction.

Such past work can be broadly characterized as being part of one of two groups, either profiling or middleware level approaches. Perhaps the closest work to our own is that of Vertical Profiling [Hauswirth et al. 2004], where an understanding of an application's performance across multiple layers of abstraction is the goal. Samples over time, from the application, middleware, kernel, and hardware performance counters can be fused into a cohesive view of the performance of an entire system stack. In Hauswirth et al. [2004] and Sweeney et al. [2004], hardware and software monitors are placed at different parts of the system which then provide performance patterns and anomalies by recording IPC (Instructions completed Per Cycle) and LSU (load/store) flushes. In Dean et al. [2004], hardware monitors are added which sample instructions and instruction pair information in order to find performance bottlenecks. The Linux system profile tool, OProfile [Levon and Elie], also samples PCs to monitor performance and whole program paths [Larus 1999]. Knowledge about different layers of the system is key to understanding complex systems. Application and middleware profiling, combined with operating system profiling, can provide deeper insights. A method for profiling operating systems based on a runtime latency distribution analysis is proposed in Joukov et al. [2006]. The main difference between our work and this prior work, is that prior profiling methods have concentrated on performance, and therefore only the most commonly occurring events are those that are reported. It is a powerful tool for optimization, but it does not attempt to indicate *how* dataflows, where specific pieces of data are used throughout the system, or how different components interact and communicate. Due to a necessarily heavyweight implementation, Dataflow Tomography is not a performance profiler nor does it serve to replace one. In fact, the two tools working together would perhaps be most useful of all, profiling to identify bottlenecks, then tracing data from those bottlenecks back to their semantic sources. Kim et al. [2009] use a similar idea to our tainting mechanism to track information throughout a system via the SeeC tool which uses per application modules and flow managers for interprocess taint propagation. Unlike SeeC we can extend tainting beyond memory to I/O devices such as network device and hard disk without the use of separate monitors. Another difference is that, unlike these profiling techniques which require adding monitors in key locations of the system to gather application and system behavior information, Dataflow Tomography separates the definition of tags (which can be placed by the developer in locations known to them) and the gathering of system information (which requires no instrumentation, or even deep knowledge, of the runtime).

While performance profiling is one use of full system tracing, it can also be used by developers to help them debug full systems. For example, deterministic replay of multiprocessor execution can be provided [Xu et al. 2003], application-level bugs can be replayed with low overheads [Narayanasamy et al. 2005], and Web applications can be profiled, monitored, and secured [Chong et al. 2007; Haeberlen et al. 2007; Kiciman and Livshits 2007]. Though tools for individual components of a system (such as the OS, middleware, and applications) are useful, the knowledge of how all these components connect and communicate could potentially aid application developers in both understanding and improving existing systems and in the design of future systems as well.

Our work is mainly motivated by the "black-box" debugging approach in distributed systems by Aguilera et al. [2003], where RPC-call based timing information is used to isolate performance bottlenecks in distributed systems. They propose two indirect techniques (such as a signal-processing inspired analysis of packet arrival and departure times) to pinpoint specific causal paths (e.g., the relationship between an incoming request and a back-end response) while treating all of the elements of the system as black-boxes (requiring no modifications to the applications or middleware). Similar interesting work, in this case, modeling workloads with instrumented components, is

proposed in Barham et al. [2004]. These approaches take a fairly coarse view of the system components in an attempt to be minimally intrusive to the software components. With Dataflow Tomography, we propose to be minimally intrusive at the other extreme, by tagging data at the finest possible level (at the byte level in hardware) so that no intrusion whatsoever at the software layers is required. To the best of our knowledge we are the first to propose dataflow analysis to aid in understanding full systems: the processor, operating system, middleware, application, disk and network, all together.

3. DATA TAGGING FOR TOMOGRAPHY IN REAL SYSTEMS

Dataflow tomography, unlike many other techniques, can easily support arbitrary levels of detail through black boxes. If there are particular abstractions which the user would prefer not to break, perhaps a particular process or shared library, this poses no problem to our tomography technique. Entire regions of execution can be treated as a black box: data-with-tags go in and data-with-tags come out. At the same time, because all the software runs through our network of virtual machines, we can drill down to an arbitrarily fine granularity, identifying those processes, modules, functions, and even instructions which touch tagged data. Because tag initiation and propagation is implemented at the ISA level, absolutely no modification to either the kernel, middleware, or applications is required.

A second significant advantage of dataflow tomography is that we can easily isolate those particular portions of the system which are relevant to specific events of interest to the user, not just those portions of the code that consume a lot of cycles. For example, if one is trying to trace a large server application that calls many different services simultaneously (perhaps across many different machines), we can insert tags on individual packets, or from within the application or its libraries directly, and identify only those portions of the system which lay in a direct chain of dependence on the event of interest. While our techniques may occasionally miss some small amount of transferred data, such as the classic examples of entangled data-control dependencies [Vachharajani et al. 2004], unlike the security applications of data information-flow tracking, users here will not likely be actively trying to hide information flows through covert channels or other adversarial means.

3.1 System-Level Overview

Because real computations often span multiple machines, we need a way of tracking data as it is both manipulated by the processor and as it is pushed through the network. The way that we accomplish this is by executing all code (including the kernel, network stack, and application) under virtualization. Associated with every byte (as opposed to every word) in the physical memory of the virtual system is a tag. As the virtualized processor operates on data, the virtual machine performs operations on the corresponding tags. As the virtualized system sends network packets, the virtual machine encapsulates those packets and sends along the corresponding tags.

Tags in this system can be treated as general purpose information rather than simple Booleans or sets of independent Booleans as in prior approaches. Four bytes of tag data per byte of physical memory is kept, which allows us to use that space as simple identifiers, as pointers to more complex metadata, or as simple sets (e.g., we often use the tag to store a range of values). In schemes where we use the contents of the tag as an identifier we will refer to those identifiers as “colors” to distinguish them from “tags” which are are the containers for that information.

QEMU [Bellard 2005], a machine emulator, provides the basic virtualization needed to implement and demonstrate our tomography tool. Qemu includes an emulator for

different CPUs (such as x86 and PowerPC), generic devices emulation for communication between the host and emulated system (block and network devices), specialized device emulation (such as a VGA display and an IDE hard disk), and low-level system control API useful for debugging. CPU instructions from the target architecture are executed on the host via dynamic binary translation. With some trivial optimizations, this simple approach is fast enough to do all of the dataflow tracking within the machine and across the network and file system, including logging all the necessary information onto the host disk, while still successfully hosting a Web server (composed of Mongrel, Ruby, the Rails framework, Perl, and MySQL) capable of serving requests at a reasonable rate.

When inserting heavy weight analysis into the system through virtualization, one concern is that by slowing down the system one may actually change the behavior of the system under examination. While in this work we do not deeply examine time-sensitive events, concentrating instead on logical information flows, the variations in timing did cause a few serious problems. The largest of those was that the bootup sequence would fail due to timeouts, a problem we circumvented by simply disabling those problematic timeouts. If one wished to more carefully examine information flow in the context of timing, one option would be to combine the work presented here with that of Gupta et al. [2006]. In that work a method of "time-dilation" is presented, in which the system is nonintrusively modified to operate on a virtual-time, running at some fraction of wall-clock time, in such a way that any necessary analysis can be hidden in the slack between the two.

3.2 Insertion, Propagation, and Extraction

Any tomography scheme we develop will rely on the ability to perform a set of basic operations over tags as follows.

Tag Insertion. Mechanisms by which an application/OS developer can associate tags, in the underlying virtual machines, with user-understandable data, in the virtual system. Examples ways in which this might occur include via the network interface, through well defined I/O operations, or by accessing particular memory locations.

Tag Propagation. the rules which govern, given operands of a specific tag, how an instruction should derive the tag of the data it produces. Propagation of tags should be consistent both within the system (through memory, registers, and potentially peripherals) and across systems in a networked environment.

Tag Extraction. the locations in the system where tags are read, and the process by which they are converted into useful views of the system. Tag extraction requires a way of mapping the hardware level tags, and the dataflows they represent, back to semantically relevant information at the application level. This, at minimum, involves not only mapping back the physical address to an application's virtual space, but also mapping the virtual addresses into the source code (if available) of the service components which make up the system.

In this section we discuss exactly how (and where) these basic primitives can be supported in a complex networked system.

Network-Level Tag Insertion and Extraction. Because the network is the system interface to both other machines and the outside world, and because there are many tools for classifying and analyzing packets, the network interface is a natural place for a user to insert and extract tag information. Our system can tag entire Ethernet frames, or even the individual bytes of a frame, with unique colors. These colors can then be tracked through the OS and various applications, back to the network card, and out of the

system over the network. To implement this we modified the NE2000 Ethernet device in QEMU to be able to monitor and propagate the tags by adding tags to its internal memory. Every time an Ethernet frame enters the Network Interface Card (NIC), the NIC local memory can generate a tag for the incoming byte. This tag is propagated to the processor when it reads from the NIC memory and writes to the processor's physical memory. Likewise, the tags can be logged before writing data out over the network. Propagating tags back out on the network requires some other modifications which are discussed in a paragraph below.

Application-Level Tag Insertion. It may be the case that a user wishes to track a specific piece of data, for example a credit card number or an argument to an RPC-call, to discover how it flows through the system and which other code operate on it. Because our system is so low level, this is somewhat more complicated than those systems where the full symbol information is known, or where special compiler support has been integrated. One way to solve this problem, and the way we have implemented it, is to ask the user to write (through a library) a range of virtual addresses that they wish to map. The library, written in C, then writes the addresses to the serial port which are then captured in the virtual machine. While this solution is difficult to implement in languages such as Java where the virtual addresses are hidden from the users, we have successfully tracked data from Ruby (which also hides virtual addresses) by creating a variable in C, registering the address of the variable with the tag tracking system, and then accessing that global variable from inside Ruby. The tag can be propagated to a Ruby variable by performing an information-preserving operation combining both the Ruby and C variables, and by storing the result in the Ruby variable (e.g., (*Rubyvar* = xor(*Cvar*,xor(*Cvar*,*Rubyvar*)). While this is admittedly not as pretty as having built-in language support (which is also possible), it does demonstrate that it is feasible to provide application level tagging even in safe languages without modification to the language or runtime.

Disk Tag Insertion. Data in real systems is often stored in and propagated through not just memory, but the file system as well. For example, take the example of a Web server with a MySQL database back-end. Information is exchanged between the two using either a local file or TCP/IP connection. In the former case, every byte that is written to the database is moved through the disk. Without disk tag tracking, crucial information would be lost. Figure 2 shows how we closed this loophole by tracking data tags at the disk interface. Since we could potentially have a very dirty disk (in which, over time, each byte of disk becomes colored in some way) tracking at the granularity of bytes can become prohibitively expensive when those colors are stored naively. As such, we use a smart compression mechanism similar to the multi-bit tag trees used in the backing store of range caching [Tiwari et al. 2008]. Rather than keeping tag information for all fixed sized (as byte or word length) data regions of an address space, the entire address space is represented in terms of consecutive address ranges and their allocated tag. Once data is transferred from memory to disk the associated tags are copied to the disk's tag map by either creating a new range or modifying previous ones.

Instruction Level Tag Insertion. The exact nature of the tag can be determined by whatever policy is implemented, but it could be as simple as a "taint bit," or something more complicated such as a unique tag based on the specific function name of the instruction executing a store to a particular memory address. This flexibility means that we may wish to take a "tag them all and sort it out later" approach, for example tagging all memory addresses with a way to identify which regions of code wrote to them last. In this case the tag needs to be generated on the fly, generated from an attribute of the

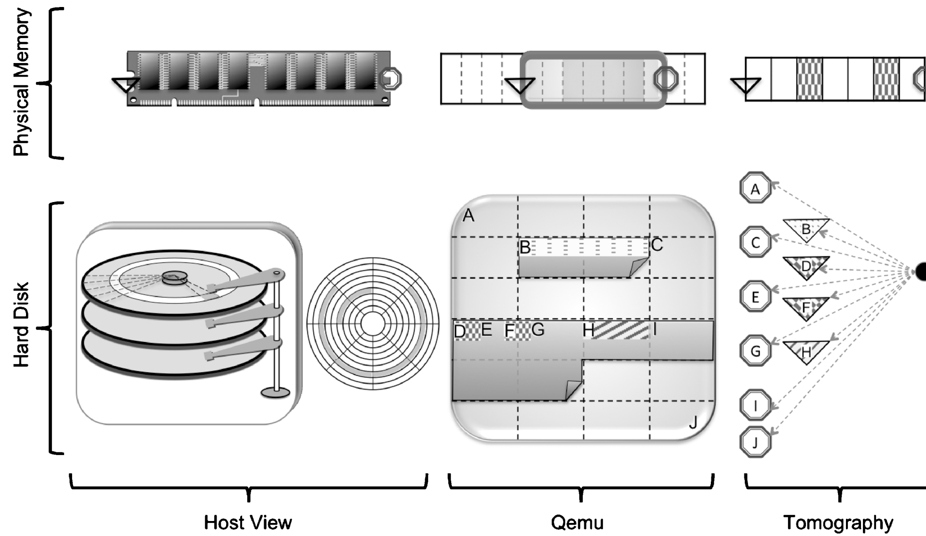

Fig. 2. An abstraction of tagging at two different locations in the system. The top row, labeled "Physical Memory," shows different representations of physical memory while the bottom row, labeled "Hard Disk" depicts how the virtual hard disk is stored. The leftmost column, labeled "Host View," gives a high level illustration of physical memory and hard disk. The middle column, labeled "Qemu," represents the implementation of physical memory and hard disk on Qemu version 0.8.2. The final column, labeled "Tomography," demonstrates the two different tagging structures we use to track data.

instruction itself, such as the region of code (function, class, or file for example) or the PID of the process that is running. Getting the PID is perhaps the most complicated aspect of this task, especially without modifying the kernel to get it. We use a combination of the CR3 register and the PID kept in the task struct to identify the PID of all instructions executed. Later we show how inserting tags on every instruction (as opposed to propagating through instructions) can be used to build a full-system dataflow graph.

Network Tag Propagation. While Section 4 concentrates on different dataflow propagation policies, propagation through the network uses the same infrastructure (because data is just transferred not transformed). The bottom of Figure 3 shows how the virtual machines communicate amongst themselves through a socket-based emulation of the Ethernet. The *eth*1 of the machine on the left running the Mongrel Web server communicates to the *eth*0 of the machine on the right running MySQLD through a socket based Ethernet emulation module which we have extended to also transfer tag data. Virtual machine communication to the outside world happens through the *tap*/*tun* virtual Ethernet device on the host operating system. The host operating system, which hosts a virtual machine with two Ethernet cards (such as the one in the left in our example), provides a *tap* device for the guest hardware to communicate to the external world, while a socket-based emulation of the Ethernet device is provided for communication among the virtual machines. In this case, when a packet from the outside world destined for the guest machine enters the *eth*0 of the host machine on the left, that frame gets routed to the *tap* interface, and the guest hardware pulls it off of its *eth*0 device. However, when the guest machine on the right wants to communicate to the external world, it writes a packet to its *eth*0 device. The socket based Ethernet emulation then reads this packet and sends it to the *eth*1 of the machine on the left (this channel is nothing but a TCP/IP socket). Once *eth*1 of the machine on the left

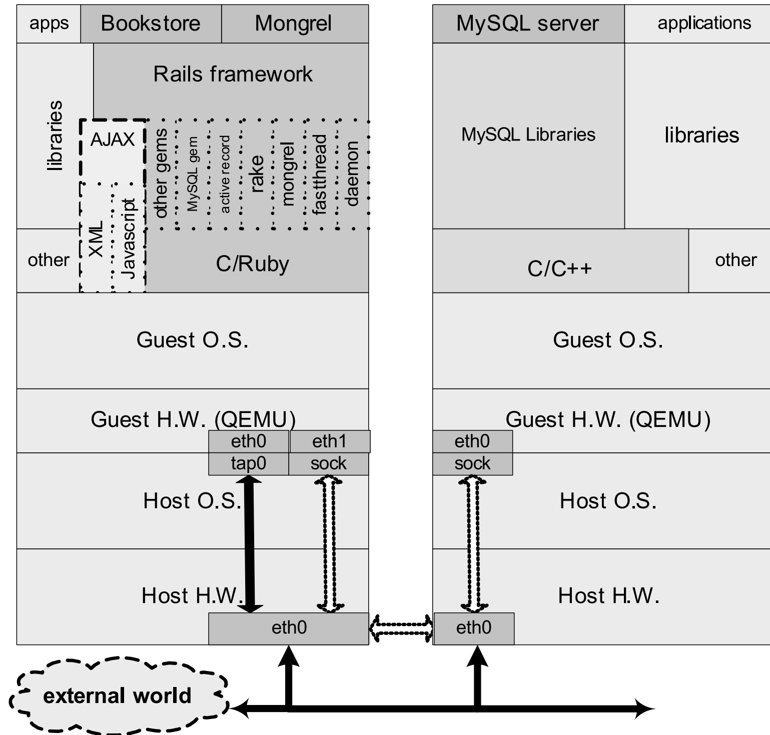

Fig. 3. Our system setup includes the host hardware, a host OS, a guest hardware (QEMU), and a guest OS. The application, in this example an online Bookstore written in Ruby and running on the Mongrel Server, can communicate in a tag-preserving way through the network to other machines running various other applications, for example to MySQLD on the machine on the right. The virtual machine on the left sees two network interfaces, one provides tagging features (*eth*1 - for traffic within the virtual network) and the other does not (*eth*0 - so that normal external communication is possible).

receives this packet, it strips the packet of all the tags associated with this packet and then forwards it to the *eth*0 on the same machine, which is in turn read on the *tap*0 of the host machine on the left and forwarded to the external world through its *eth*0.

Disk Tag Propagation. When it comes time to read from the file system, we use the reverse transfer of tags from disk to memory to continue the uninterrupted flow of tags within the system. Once the operating system wants to write a file to disk, the requested data is moved from physical memory to disk one page size at a time, where the size of the page depends on the hardware being emulated. At boot time Qemu emulates an IDE hard disk image by identifying the geometry of the disk in terms of cylinders, heads and sectors. Data is written to the guest image sitting on the host machine one sector at a time. Once the IDE device driver is called to transfer a data to disk, it takes a specified starting address of physical memory and copies information to sectors. At the same time, the system grabs physical memory tags and updated the tag map for the disk locations that are accessed (which could span from a single byte to multiple sectors). During bootup, when the kernel attempts to mount the file system, we disable disk tracking to address two issues: To prevent an explosion of tags and to avoid time-outs that would occur when mounting relatively large filesystems. Tag retrieval would compounds the inherently slow process of accessing data off disk causing the kernel to believe there has been a faulty disk access.

Tag Extraction via Reverse Address Mapping. Once the processors and the network cards involved in our distributed system setup are capable of inserting tags at the behest of the application/OS developer and propagating them within the system, the final step is to be able to make sense of the resulting tags. Here, because we need to map back to something that a user can understand, we do need to be at least aware of the applications. Systems which implement tagging from the application level and instrument the kernel, middleware, applications have an easier time making sense of the tags, but come with the disadvantage of having to modify the kernel, libraries, middleware, and sometimes even the application. To handle all of this interprocess communication and data sharing that happens in real systems, tagging needs to happen on physical memory, but end users will only be able to understand virtual address (or more specifically they understand structures that are mapped into to virtual addresses). Instead of modifying the kernel, we probe the currently executing process's *task_struct* in the Linux operating system (at the VM level) and map the physical memory addresses to the corresponding virtual address space of a particular process. We then use information available from the `/proc/<pid>/maps` and the `objdump` outputs for the component systems, along with the kernel symbol table, to map hardware level tag inferences back to the application.

3.3 Tag Framework

Figure 2 shows the conceptual view of our Tomography implementation. In this figure we have shown two locations where tags are inserted and propogated, then represented each location at three different levels of abstraction.

Host View. Data is handled with different granularities in physical memory and hard disk. Let's assume that the Host is manipulating data on RAM with a physical starting address (shown by the upside down triangle in the left most end of RAM diagram) and an ending address (marked by the octagonal shape at the rightmost end). At the physical memory level a varying number of bytes can be read or written at a time. On the other hand, for accessing disk, the operating system first determines disk geometry in terms of platters, heads, number of sectors, etc. and then data is transferred in sector units (typically 512 bytes). In the example figure, at the Hard Disk location, Host View level, the Host sees a hard disk composed of three platters. The diagram highlights one track with a white concentric circle and a single sector being accessed via the disk read/write head. To the right we see a top view of a platter with multiple sectors filled with data. Let's assume the outermost 6th consecutive sectors is one file and the inner 2nd consecutive sectors is another file in the file system.

Qemu Implementation. On Qemu a large buffer stores memory for different devices such as the VGA card and RAM. The section emphasized with an outlined rounded green box represents a subset of the buffer that would be used as the emulated RAM. The four darkened cells represent data written to memory. On the other hand, the emulated disk, which is a single large bootable image file on Host, is represented by a 2D table where each cell represents a sector. Continuing with the Host View example, we've represented the same two files written to the disk image. For simplicity, a set of consecutively filled cells represents a file.

Tomography Tag Structures. A simple one dimensional array holds tags for physical memory, and a tree structure (root marked by a solid black circle) manages tags for the hard disk. Each byte in physical memory has a mirror entry in the array tag structure. Following the example of data written to RAM, of the four pieces of data in physical memory, two entries have a tag associated with them (shown by the filled checkered

boxes). As data is block copied from memory to disk, the tags are compressed and inserted in the hard disk tag tree. In the example, the first tagged data from memory starts at address D and ends in E while the second starts at address F and ends in address G.

3.4 System Setup

Figure 3 provides an overview of the layers involved, and the modifications to the virtual machine required to build our Dataflow Tomography tool. The topmost layers show our example application layers which are unaware of the underlying dataflow tracking. In our specific test-setup, Mongrel [Shaw] serves an online bookstore application while a MySQLD back-end runs on another machine across the network. Both the libraries which drive our application, and the mongrel server utilities itself, are based on a combination of C and Ruby, and tags flow seamlessly between them. The bookstore application and Mongrel use the Rails framework, which is composed of various libraries, termed "gems," based again on C and Ruby. AJAX is used for "Shopping Cart" management features. In other experiments we make use of an Apache Web server and, in addition, for illustrative purposes, we use a simple "Hello World" application that just receives a request (via TCP) and responds with a single packet. In Section 5 we further discuss how various real world applications such as compilation and database manipulation performed on a single machine.

4. TOMOGRAPHY SCENARIOS

Using the infrastructure described in Section 3, we have built three example systems to help us map various aspects of the systems described in Section 3.4.

4.1 Use Case 1: Tag-and-Release

Tag-and-Release is the set of policies most analogous to security or privacy information flow tracking. Specific pieces of data, with known semantic relevance (perhaps a global variable, packet field, or a shared memory buffer) are tagged and the system is allowed to run for some amount of time. During this time that tag is allowed to propagate throughout the system according to the specific policy. Most past papers assume tags are booleans and propagation is a simple logic operation such as *and*, but we will describe how more complex functions can be useful in Section 4. Extraction occurs after the execution completes. The system is analyzed to determine where the tagged data ended up, which inputs and outputs were affected by the tagged data, and which code (processes, modules, procedures, etc.) touched each particular tag. Multiple tags from multiple sources can of course be in flight at the same time, but the key idea is that tags are inserted at known places of interest and knowledge about the system is gained from the resulting distribution of tags.

4.2 Tag-and-Release Example: Query Data Mapping

Consider a developer asked to change the way credit card data is handled for an online bookstore. One of the first things to consider is where and how the credit card information, entered by a remote user, flows in her system (typically a networked system of several machines). A Tag-and-Release approach lets us track this information within a single machine (from packet in, to packet out), and also across the network wherever the credit card information flows, as long as it is within the virtual system. To solve such a problem, we need to choose a method of tag initiation, tag propagation, and tag extraction.

Tag Insertion. For this specific problem, there are two logical channels that could initiate tagging. Tagging could be initiated at the network interface of a machine or by the application itself using the serial port calls. If the data occupies a known location within a packet, for example if it is encoded into the packet as a structured field, it can easily be identified and tagged. If the data is part of complicated, yet widely adopted, network protocol, then a call to any number of packet classification tools could yield the location in the packet of the data of interest. However, if that is not possible, a third option exists; run the client in the virtual environment along with the server. This allows us to tag the data at the client where its location in storage is precisely known, and to propagate those tags through the network to the server system, and eventually even back to the client.

Tag Propagation. In this scenario, the tag propagation rules are straight forward and very similar to the many related ISA-level dataflow tracking techniques built for security. If either input operand is tagged, then the output should be tagged. Multiple colors can be tracked at the same time, relating to different data of interest (e.g., the password and credit card number). Here, because the tags are fairly sparse in the system, tag collisions (i.e., where operands both have different colors) are not very likely to occur and can be resolved through either randomization or priority. The major difference with our policy from past security work is that we need to track the use of these tags back to something of meaning to the user. A simple allow/deny decision is not what we are after. To solve this problem, we can track the code that operates on tainted data. Every instruction that touches a tagged data item become *stained*. Stained is not just another way of saying tainted or tagged—code that is stained was never written with tainted data, rather it has just touched tainted data. Of course, one of the complications is that tags operate on physical address, but the physical address of code is not something a user can readily use. Instead, we maintain our tags on physical data but our stained-code map on tuples of $< virtualaddress, PID, CpuID >$.

Tag Extraction. When our system has finished executing, we end up with a list of tuples (which need to be stored in a compressed manner) describing all of the tainted data. We then can use the symbol tables of the applications and kernel to map the address back to user understandable information.

As a way to explain the Tag-and-Release approach, we ran our simple "Hello World" network server on our tomography tool and tracked how the "Hello World" string flowed through the system and across the network to a remote client. Figure 4 shows the names of the functions that touched tagged data (in our case, the "Hello World" string). This representation shows the set of the functions that used the tagged data on the y-axis (either directly, or indirectly on the data that was derived from tagged data) and the x-axis represents execution order. The figure shows that a main function initiates the tagging of the string "Hello World" which then flows out on the NE2000 device, passing through a TCP/IP socket. It can also be seen that the interrupt processing routines operate on the tagged data, due to context save/restore process which now is forced to operate on tagged data in the process's address space.

It is worth noting that this data goes beyond what can be achieved through simple call tracing. A call trace will return a sequence of functions executed over time, but the graph in Figure 4 identifies the subset of functions which actually use/propagate the data we are examining (in this case a simple string, but it could even be a credit card number, a particular object, a password, or even a "cookie" as we show later in the paper). For this simple server example, at least 116 different functions are invoked, yet less than 40 of them actually touch the data. Of course, such an ability is even more critical when you have multiple services, scripts, an OS, programming languages, and

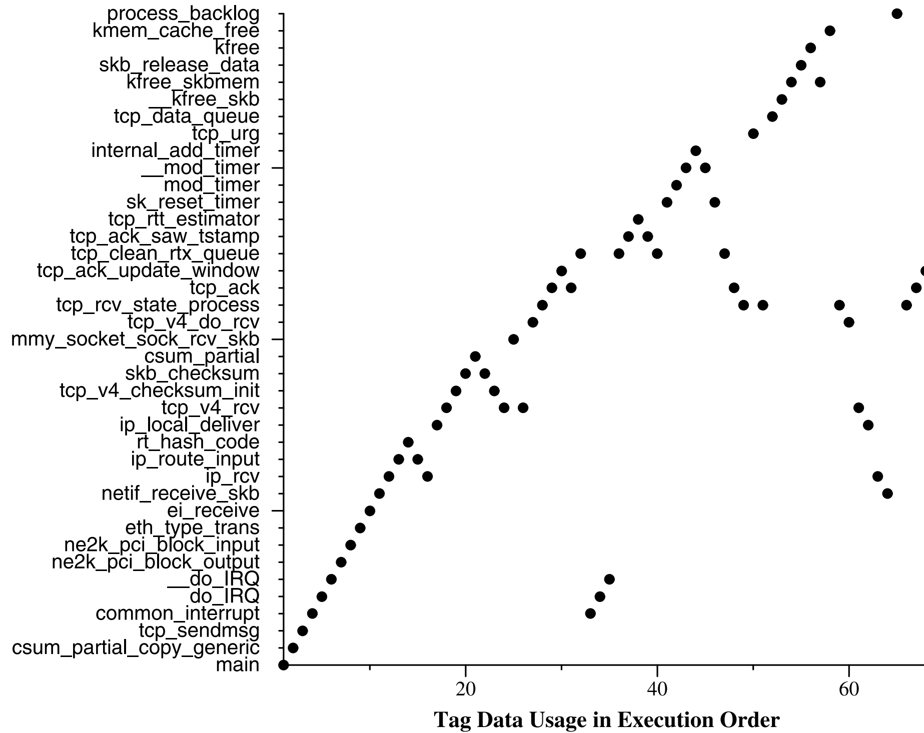

Fig. 4. Execution of functions in the order of dataflow. At step 0, the "Hello World" string is tagged by the server application (by communicating to the Tomography tool through the serial port). As the server executes, and as functions touch and spread the tagged data, a list of relevant procedures is created. The y-axis shows the procedures names of those functions touching the tagged data as the server executes.

even machines talking to one another. Later, in Figure 7, we will revisit this idea for a full Web server (running AJAX, Ruby, Rails, etc.) and show that even tracking three different dataflow slices, less that 60 different functions touch the data of interest (most of which are TCP functions already visible in our simple "Hello World" example). Dataflow tomography allows us to slice through the otherwise unwieldy number of functions invoked during, even a single second, of Web server operation.

4.3 Use Case 2: Flow-Source Tagging

Flow-Source Tagging. looks to answer questions about where the data comes from rather than where the data is going. In this set of policies, tags are inserted throughout the system and across its inputs to ensure that the majority of information flowing through system carries some useful tag. Extraction occurs once a point of interest has been reached (a line of code, a packet being written, etc.), at which time the tag for the data in question can be queried to learn some information about where the data came from. For instance, one useful but difficult to implement example would be to give every byte of memory and every byte of input a unique tag, and to track those tags through the system. However, because there are so many tags in flight in the system, one of the key complications is how tags are merged. For example, how do you determine the tag of an add instruction with two operands of different tag types? Either the tags have to be unbounded sets, or some information must be lost in the merging.

4.4 Flow-Source Tagging Example: Byte-level Packet Dependency Analysis

As systems are built of abstraction over abstraction, it is inevitable that some of the components (for example third party libraries, shrink-wrapped products, even whole machines) will essentially need to be "black boxes." Because Dataflow Tomography can run underneath these black boxes and requires no direct support from the system under test, it can be used to recognize the relationships between the data going in-to and coming out-from from these black boxes. In most real world scenarios, the information that flows into a system is combined in a very complex way before resulting in an output, and teasing apart which incoming information influenced which outgoing information allows for a much deeper understanding of the resulting system. For example, if we treat an entire machine as a black-box, we may be able to discover the relationship between outgoing packets, and the input packets that caused them (without relying on any preexisting application-level knowledge). In fact, we show below that because we track dependencies at such a fine granularity, Dataflow Tomography can discover not only causality relationships between packets, but even the internal structure of arbitrary packets (both input and output) based on how the data in the packets is used.

The main idea behind this approach is to describe the color of the bytes of outgoing packets as some combination of the colors of the inputs.

Tag Insertion. Flow-Source Tagging works by tagging large amounts of relevant data, for example every byte of incoming traffic. We tag every byte that arrivies off the virtual Ethernet card with a unique color so that we can identify which outgoing network bytes were dependent on which of the incoming bytes.

Tag Propagation. Because of the large number of tags in flight (on the order of many thousands or more), tag propagation becomes very important. The goal is to be able to look at the resulting tag and identify the source of the path from which it came. However, if two operands have different colors, and hence different sources, then how are we to combine them into a new tag? The output of that operation came equally from both sources. To solve this problem, we actually divide the tag into two parts, and use it to store a range of integers (or a spectrum of colors if you prefer to stick with the color analogy). This merging function is lossy, we only know roughly where the original sources of particular piece of data are, but as long as there is some spatial locality this can provide a great deal of information.

Tag Extraction. Since we tag every byte of the incoming network information (specifically, we tag Ethernet frames at the Network Interface Card because the virtual machine has no notion of higher level packets) with a unique color and observe how operations spread these colors within a system, and we can observe the resulting combination of colors from an outgoing frame. The output packet actually has, for each byte, the range of data from the incoming packets that it was dependent on. We can examine all of the outgoing packets to find the ranges in the incoming packets that turned out to be very important; tracing the bytes back to their flow-source. It turns out that very often there are only a few sets of ranges that occur in the output packets, and we can draw them by assigning each of them a unique real color.[1] In other words, if we find that bytes 0–32 for many of the outgoing frames are influenced by bytes 24–25, we would assign the incoming range 24–25 a color (gray in this case) and draw both 24–25 of the incoming frames gray, and all the bytes of the outgoing frame influenced by 24–25 in gray as well. While there are many possible problems in rendering

[1]Here we mean a physical color as drawn in a visualization, not to be confused with dataflow tag colors.

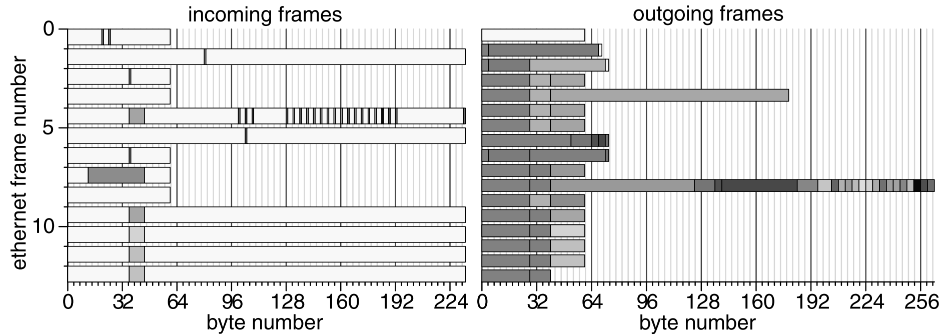

Fig. 5. The above figure shows the incoming Ethernet frames with those regions that were discovered to flow from the input to the output for the simple wget experiment. The colors show the mapping between bytes in the incoming frames and they use in computing the data used in the outgoing frames.

the ranges in this way (in particular complex overlapping ranges), none of the data we examined exhibited any of these problematic behaviors. In fact, instead we found surprisingly clean fields are discovered in the packets through this fairly simple analysis, with no use of any higher level knowledge.

To further explain Flow-Source Tagging, we use two example packet flows. First, a wget which sends a HTTP request for a Web page and receives the Web page in response. In Figure 5 (best seen in color) we find how the outgoing bytes in an Ethernet frame are influenced by a few of the key incoming bytes. In Figure 5, the y-axis shows the incoming Ethernet frames on the left and outgoing Ethernet frames on the right, while the x-axis shows the byte numbers. The color of each block in an outgoing frame (as described above) shows the bytes of the outgoing frame which are tagged by a particular incoming byte (shown by the same color in an incoming frame). White boxes in the incoming packets of visualization are data that do not flow to the output packets. For example, we can find how an incoming byte in the very first frame (colored gray) influenced almost every outgoing frame (gray colored boxes in the initial bytes of the outgoing frame). A similar result is shown in Figure 6, this time for our sample ruby Web application. While finding these fields alone has interesting applications in protocol and network traffic analysis, as an application developer, once one discovers which bytes are important, a natural question is to want to know what they do and which functions operate on them. The actual semantics of those bytes can then be found in application itself, through a Tag-and-Release marking each of those fields. Figure 7 shows exactly that, with bytes corresponding to the header, cookie, and user data (a more detailed explanation can be found in the figure caption). Although all three pieces of data appear at the network IP+Net (right most) y-axis, only the last two reach the application. The information is logged to the console and the console writes are clearly visible around 130 and 160. This figure is furthermore a good demonstration of the "black-box" abilities of Dataflow Tomography; if the application developer prefers to be unaware of the underlying software stacks, the data can easily be filtered to show only those bytes which actually come in from the network and are directly used by the application. In fact, if we mark the stained code with the *ranges* of incoming bytes that they operate on as well, we can compute both the flow-source and map those flow-sources to code (as shown in Figure 7) in a single pass.

4.5 Use Case 3: Confluence Tagging

Confluence Tagging unlike the other techniques, seeks to uncover, not the final location of the tags, but rather the points in the system where tags collide. If the initial

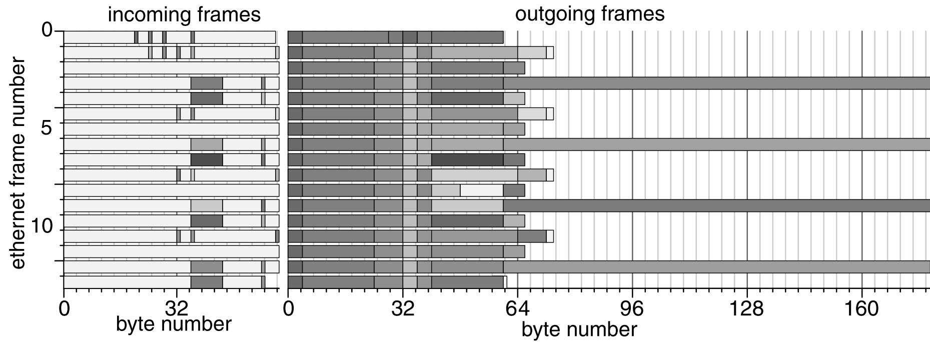

Fig. 6. A simple Web server running on Mongrel/Ruby shows the that there is a lot of correlation between the incoming and outgoing Ethernet frames. As in Figure 5, the colors of the outgoing frames are influenced by the colors of the incoming frames.

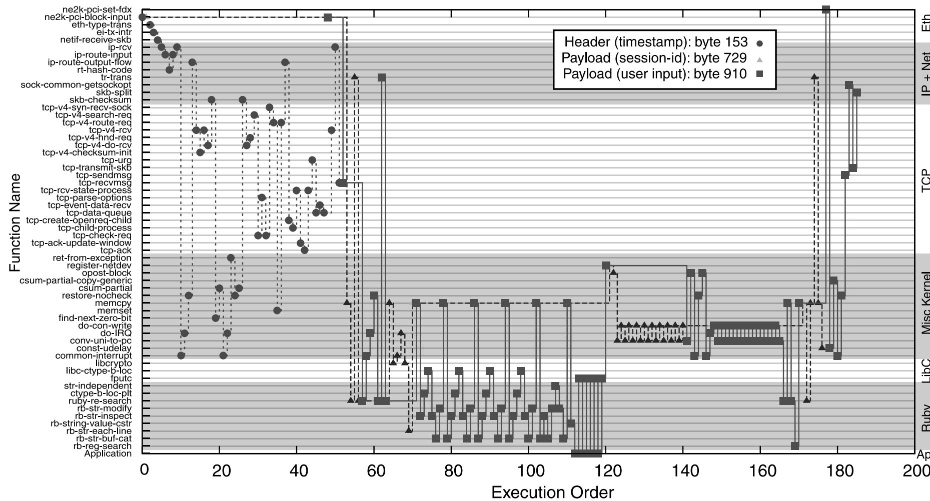

Fig. 7. Execution of functions in the order of dataflow for three different bytes of interest. The experiment shows a further step beyond the initial packet analysis from Figure 6. Once fields are identified through flow-source tracking, the semantics of those fields can be discovered by tag-and-release (done during a single combined run), and in this case the three different fields are traced. In this case one byte corresponded to the time stamp in the header (and which shows up in a variety of TCP related invocations), one byte corresponded to the session-id cookie (and is used in Ruby and Rails for session management), while the last byte actually corresponds to user data entered into a field on the Web page (this byte makes it all the way through the software stacks to finally appear at the application. The information is logged to the console and the console writes are clearly visible around 130 and 160. This figure is furthermore a good demonstration of the "black-box" abilities of Dataflow Tomography; if the application developer prefers to be unaware of the underlying software stacks, the data can easily be filtered to show only those bytes which actually come in from the network and are directly used by the application.

distribution of tags is set up such that the flows have some end meaning to the user (as in Flow-Source Tagging), the points where multiple tags of different color are being combined, overwritten, or generally transformed together, are often interface regions in the system. For example, communication through shared memory can be discovered by giving each process a unique color and identifying when a process of one color is reading the data of another. The goal being to extract the exact place where data

is moving across boundaries, where the boundaries themselves are stored dynamically with the data itself.

4.6 Confluence Tagging Example: Identifying Cross-Abstraction Communication

While assigning arbitrary colors as unique tags in the Tag-and-Release and Flow-Source tagging approaches helps uncover where a tag came from for a piece of data and who used that data, another way of using tagging is to try and find how different tags interact within the system. We show how Confluence Tagging can be used to identify all interfunction, intermodule, interprocess, and interprocessor communication.

Tag Insertion. As explained in Section 3, we map the physical address which is tagged and the Program Counter which is "tagging" a particular memory address all the way back to its process, module, application, or even the function that is executing on the "tagged" information. With Confluence tagging, we can now find out which functions or processes exchanged tagged data. To accomplish this, each store instruction is capable of generating its own tag when it stores. In addition, we need to insert some nonzero seed tag, which can come either from the network or the application.

Tag Propagation. To keep the advantage of dataflow tracking, we divide the tags into 0 and nonzero. If the tag of both of the operands of an instruction is zero, then the output tag is also zero. However, if either of the operands of an instruction is nonzero, then the nonzero tag is propagated. If a store instruction executes and is attempting store data that is tagged as nonzero, then the store instruction writes a tag that encodes its function identifier (a unique number given to each function in the system), not the tag value it was passed.

Tag Extraction. Information is gained about the system every time a load instruction reads a data value with a tag that is different than its own. When this happens, we know that data has flowed between functions (or threads, processes, or any other distinguishing feature we care to analyze). In an analogous way we can track the flow of data between modules of code (however they might be defined), processes, and even processors.

To test this method we used Apache with a CGI-Perl interprocess dataflow as an example system, in addition to our simple "Hello World" server. Figure 8 shows how the tagged dataflow in the simple server program. Each large rectangle represents a process and in this case the dataflow between the server and the kernel is shown. Every smaller rectangle within the gray box represents a module, such as the TCP module, IP module, NE2000 network interface card module, etc.[2] The black dots represent an incoming channel into the module and a black circle represents an outgoing channel. Developers interested in examining finer details can zoom in on each of the smaller rectangles to see how tagged dataflowed within modules. In addition, full path information is shown with colors, so that the specific modules that talk to one another can be identified (this is not shown here). We can then observe through these graphs how tagged dataflowed within the system. We conducted two experiments to evaluate the ability to map communication in a system. First, an entire incoming packet was tagged at the NE2000 and we observed how the data and the header flowed in the system. Second, we tagged just a part of the payload and observed the different path the tagged information now flowed. Analyzing how the dataflows across different processes in a scenario such as Apache and CGI/Perl is much more interesting and obviously more complex. Figure 9 is a representation of such a dataflow as captured by

[2]This is based on the procedure name.

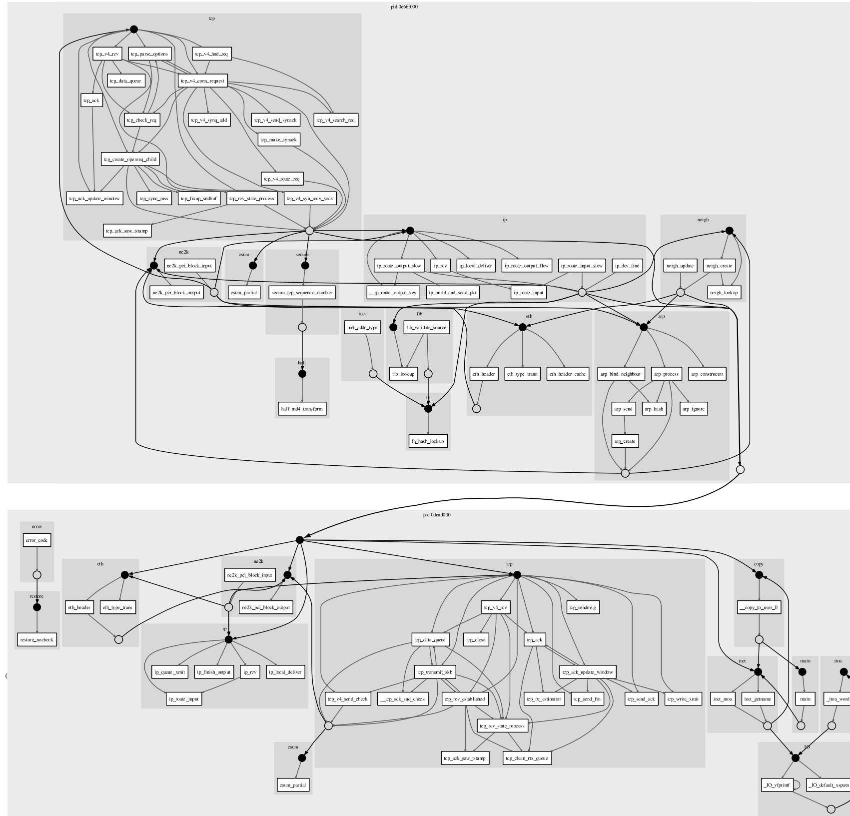

Fig. 8. A Confluence Tagging based dataflow diagram is shown for a simple network server program. The outermost rectangle represents one process, and the next inner rectangles represents modules within the process, and next smaller rectangles represent the individual functions which have exchanged tagged data.

our tomography tool. These visualizations are just one way of showing the data that can be gathered, and we are currently experimenting with other ways of presenting the data to users. The advantage that the dataflow approach has here is the ability to abstract-away large portions of the graph. For example, the common TCP and IP handling routines could easily be hidden and treated as simple conduits for information.

5. EVALUATION

In addition to evaluating our Tomography tool by the kind of information that could be gathered as shown in Figures 6, 4, and 9, we want to get a picture of the costs associated with tracking large amount of information. This is not a performance study but rather evaluates the effect that our tool has on real world application.

5.1 Experimental Setup

We have executed various applications such as compression, compilation and database access on a single processor emulated machine and native host machine. The host machine is a i686-smp CORE 2 Duo running CentOS Enterprise Linux kernel version 2.6.9–55 with 2 GB of physical memory. To reduce the amount of space required to store tag information for memory, which not only includes the emulated physical memory but for example the emulated graphics card, we limited the maximum amount of

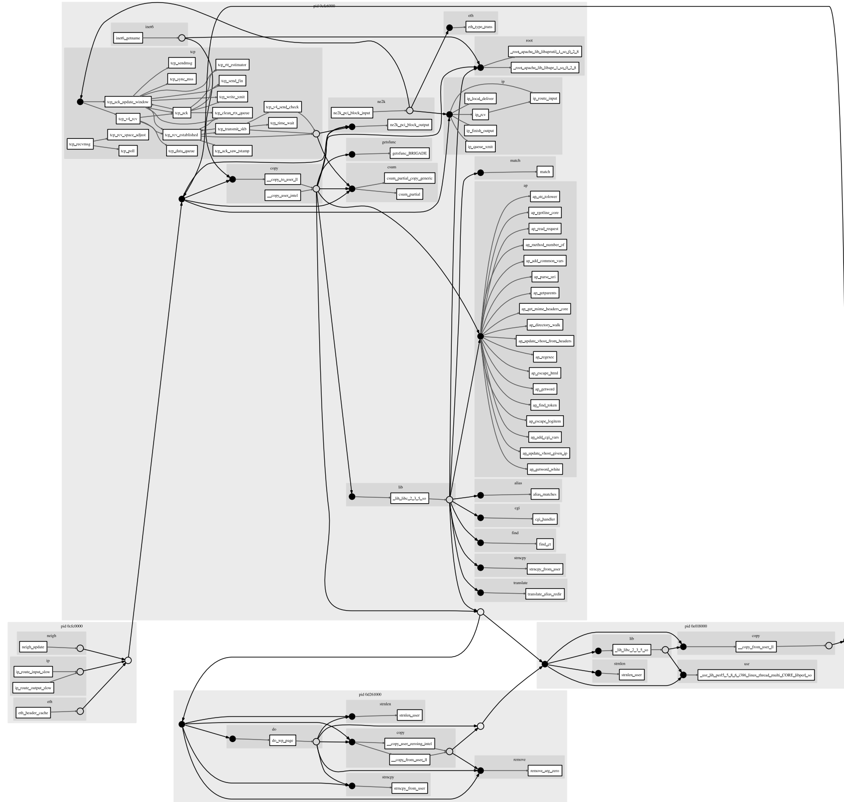

Fig. 9. Dataflow diagram for Apache Web server handling CGI requests. The CGI request processing is handled as a separate process by Perl interpreter. The dataflow between multiple processes including Apache Web server, Perl, and kernel process is shown and each can be further zoomed in to understand the internal dataflow within process or even within a module in the process.

memory that an application consumed. Both guest and host machines were given 256 MB of physical memory to execute the given benchmark.

Figures 10 and 11 show the impact of our prototype Tomography tool as compared to running the same benchmark on native host and different versions of Qemu. Each set of bars represent how an application performs on each machine in relation to the execution time on the host machine. We ran each application multiple times with the execution runtime varying from less than one to two percent of each other. As the runtimes were within a close range of each other, we used the average for our evaluation. Since we are more interested in how creating, transforming and transferring tags effects the performance of the system, rather than looking at the raw execution run times, we focused on the comparative slowdown of a system running varying tag policies (i.e., no-tag, memory+network tagging, disk+memory+network tagging) and memory policies (i.e., user mode or mixture of kernel/user execution). The average runtime of a test set (such as decompressing a 17MB machine) was then divided by the the average running time of the application on host. In the case of host machine slowdown, the bar is always 1 unit. Therefore, using comparative slowdown we are able to see a clearer picture of how tag tracking effects various applications, whether they be computationally expensive, or I/O expensive.

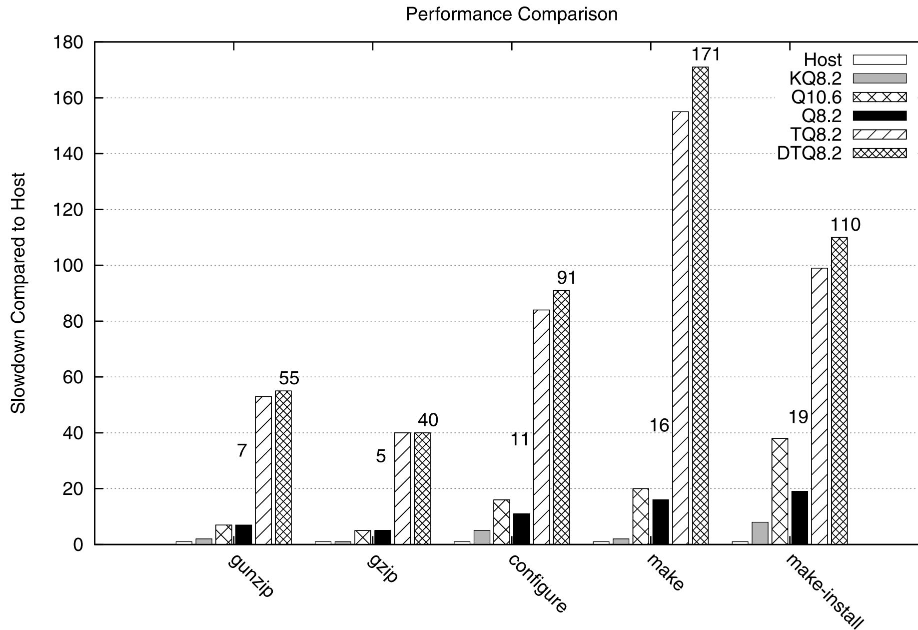

Fig. 10. Represents the impact of our prototype tagging mechanism on the performance of an application. Performance is represented in terms of how many times slower an application runs on a virtual machine as compared to on the native machine, that is, Host. The different machines are: KQ8.2, accelerated Qemu emulator using Linux kernel module; Q10.6, current unmodified version of Qemu; Q8.2, unmodified version of Qemu we used as the base of our Tomography tool; TQ8.2, modified version of Qemu 0.8.2 with tagging tracking implemented on physical memory and network interface; DTQ8.2, extended version of TQ8.2 with tagging mechanism enabled on hard disk. The different applications are: `gunzip` on Python-2.6.2.tar, gzip on Python-2.6.2.tar, compiling (`configure`, `make`, `make install`) the Python-2.6.2 directory.

Since our prototype Tomography tool is implemented on Qemu version 0.8.2 user mode, we've also included the current 0.10.6 version of Qemu in our performance results. In addition, as most Virtual Machine analysis techniques focus on comparing the fastest version of a VM with all its bells and whistles against other VMs for performance evaluation, we've also included a comparison of the runtime of an application on accelerated Qemu referred to as KQEMU. As shown by the second leftmost bar in each experimental set, KQEMU is able to achieve runtime speeds similar to that of native host on applications such as `zip`, `create table` and `delete data`. This is due to the fact that KQemu provides kernel module support. The greatest slowdown of 11.38 times was incurred at Wisconsin's MySQL `create table` benchmark where the principle goal is storing data to disk for later access.

To accurately compare the performance impact our Tomography tool has on the runtime of an application, we needed to keep in mind how time is observed in the execution environment. Not only is each OS's notion of the progression of time system dependent, but this difference is much more prominent when comparing VMs. Therefore, we used the elapsed time returned by the host machine's OS as the stable measuring scale and recorded the execution time of an application from outside the emulator it was running on. This was done using a simple UDP server to keep track of the start and stop time of each experiment. To reduce delay noise arising from message passing over the Ethernet/tap interface (shown on Figure 3), we calculated the average network delay and subtracted that from the total time recorded at the end of an execution by the host's OS.

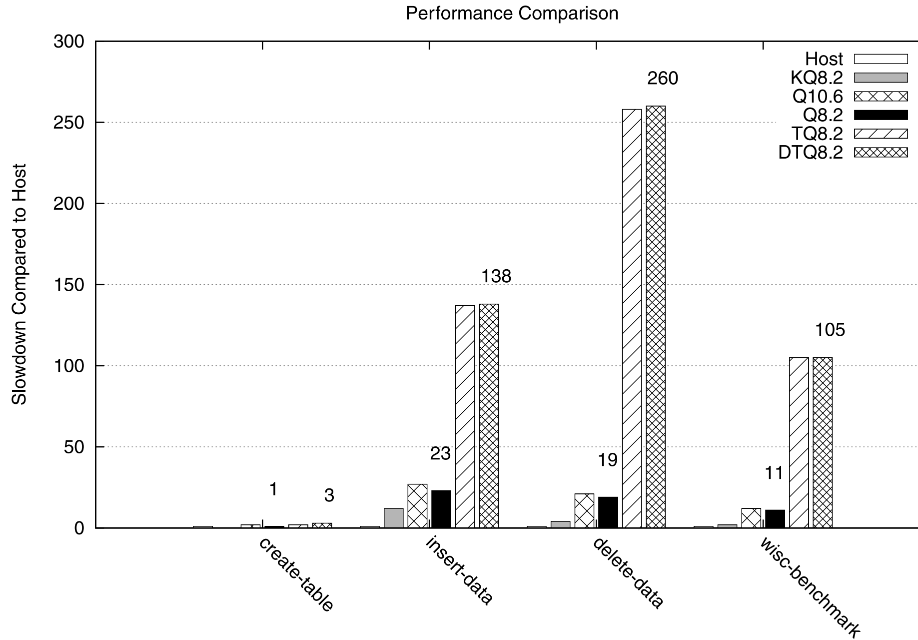

Fig. 11. Similar to Figure 10, here we show performance slowdown in terms of the number of times slower an application executes on a Qemu emulator in comparison to when executing on the Host machine. This set of experiments is based on the Wisconsin Benchmark Query Suite where we've focused on manipulating data in a MySQL database. `insert-data`, where data is read from two files with a combined size of 777 KB and written to about 30,000 rows in the database, shows very similar slowdowns when executing on TQ8.2 versus DTQ8.2.

5.2 Impact of Tracking Tags

In Figure 10 two of the bars in each application's run set are highlighted with a numerical value above. The leftmost digit above the solid black bar represents the runtime of the application on unmodified Qemu version 0.8.2 (labeled Q8.2). The rightmost bar represents the runtime of the application with tags flowing in different layers of the system through network, memory and hard disk. The first level of slowdown comes from emulation. For example, slowdown on KQ8.2 ranges from about 2 times on `gunzip` to about 147 times when running `make`. A much more noticeable slowdown appears when kernel acceleration is not enabled in Q8.2. To compare the effect that Qemu's more recent qcow2 image format has on disk access, we ran the same benchmarks on Q10.6. Overall, the Q10.6 has the same or slightly higher slowdown than on the Q8.2 emulator. Another key comparison we'll delve into next is the difference between enabling or disabling tracking tags to disk. DTQ8.2 has slowdowns ranging from less than 1 to about 10 percent that of executing the application on Q8.2.

As shown, an application's slowdown ranges from 1 to 23 times slower on Q8.2 as compared to host where the mean slowdown is 11 times. On DTQ8.2 an application can see a slowdown ranging from 3 times slower to 260 times slower with the mean of 105 slowdown. This means that the price of running on a simple Qemu emulator is a 10x slowdown and providing complete tagging mechanism incurs another 10x slowdown.

Applications that are computationally intensive exhibit less of a slowdown as compared to those that require a greater number of reads and writes, whether they are to physical memory or to permanent disk. Let's take a look at the first set of

experiments shown in Figure 10 where we've compressed, decompressed and compiled GNU's Python 2.6.2 on our emulated x86. On the machines were we are tracking tags, we've tainted the 104 KB file `configure.in` which is critical to the compilation of the source code. We've used the 66 MB tar ball directory in our leftmost two experiments. Here we see that running GNU's zip (`gzip`) on DTQ8.2 exhibited a slowdown of 9 times that of the execution time on Q8.2. In comparison, decompressing the file back to it's original size ran 7.8 times slower on DTQ8.2 than on Q8.2. This increased cost is not due to the larger file size as evident when comparing the TQ8.2 and DTQ8.2 bars which have negligibly different slowdowns. The difference between tracking tags during large database transactions on the two machines was an increase of about a 1% slowdown on DTQ8.2 as compared to the slowdown on TQ8.2 (i.e., 138.25 versus 137.21 times slowdown). Other database operations such as `create-table`, `delete-data` and `wisc-benchmark` have similar slowdown ratios between TQ8.2 and DTQ8.2. Instead the slowdown is due to the increased number of loads and stores required to determine the patterns for encoding information. On the other hand, the performance of decompressing the file was significantly affected by disk tracking, resulting in DTQ8.2 runtime being 2.5 slower than TQ8.2. On the same note, during compilation `configure` and `make install` where there are a limited number of dependencies that could be affected by tainted data, have a much smaller slowdown of 8.3 and 5.8 times, respectively. In contrast, in `make` where the bulk of the code is being modified, tracking tainted information results in a slowdown of over 10 times that of Q8.2. The second set of experiments shown in Figure 11 is based on the Wisconsin Benchmark Query Suite for relational databases which measures performance of a database system by performing relational operations such as selection, projection, joins, appending, deleting on synthetic data [Bitton and Turbyfill 1988]. The interesting result here is the difference between the cost of modifying the database when adding versus removing data. Tracking data being deleted from the MySQL database impacted the system twice as much with a slowdown of 13.7 times in comparison to a slowdown of 6 times when adding data to database tables. Inserted data was simply appended to the end of the table. However, when deleting data the entire table was searched for any data that matched a query request and its duplicates. Remember that our Tomography tool works by tracking information by tag propagation. That means that if the query is tainted, any data that it transforms will also be tainted. Due to heavy tag propagation deleting data from disk resulted in significant slowdown on TQ8.2 and DTQ8.2. In general, the impact on an application's performance is dependent on the amount of tracking versus computation on information.

6. CONCLUSIONS

While there are many advantages to dataflow tomography, there are certainly many open problems remaining.

First off, the method is inherently heavyweight compared to other approaches, in both memory and time. To be a useful tool in the life cycle of a system, methods will be needed to speed the analysis. While there is certainly a convenience issue associated with making the analysis run more quickly, the bigger problem is avoiding the kernel and application "time-outs" which, if tripped, can completely break the system (making network communication impossible or resulting in hard disk boot up time outs due to slow disk access for example). Our experience indicates that tagging within a single virtual machine is rarely problematic, and that the bigger problem is in the encapsulation and transport of tags between distributed virtual machines and disk. The scalability of such an approach as we increase the number of nodes beyond two is certainly a question.

Second, a problem shared by this and most other ISA-level dataflow tracking approaches, is how dataflow and control-flow can be integrated into a cohesive and *complete* view of the system behavior. It is well known that an adversary may construct a "tag scrubber" that uses control-flow to avoid direct data dependencies between two data-dependent variables, but these examples also appear in non adversarial conditions as well. For example, a particular byte may trigger a function call or interrupt, while the actual value of the byte might never be used arithmetically within the resulting activity. Methods for quantifying and eventually capturing such mixed data-control dependencies (preferably capable of handling interprocess and interlanguage communication) are needed.

Third, while we have shown that full system information tracking at a fine granularity is feasible, the result is significant costs in performance. The cost of tracking at the instruction granularity is due to the amount of computation and storage necessary to transform multibit tags. These access patterns do not tend to play nicely in the cache, and while there is room for some performance gains such as by providing a compact storage for physical memory tags, ultimately the challenge is in how to efficiently manipulate large tags (such as the 32 bit values we've used) while still providing the level of detail necessary to accurately copy, merge and remove tags flowing throughout the system.

Finally, due to the magnitude of data available from Dataflow Tomography, better methods of visualization are very clearly needed. Our confluence tracking graphs look like "rats' nests" in large part because there are no visualization techniques available that are able to naturally handle both the idea of hierarchy *and* a high degree of connectivity. Both of these (from our experience) are absolutely required to make sense of the dataflowing through these huge complex systems.

While we have attempted to describe the challenges remaining in this line of research, we believe that as systems are increasingly developed as compositions of complex interacting services, the need to visualize and understand these compositions will continue to grow in importance.

Along these lines, Dataflow Tomography has several advantages, and we have developed both an intellectual framework for developing such systems as well as working prototypes of several different tomographic policies. The first class of policies, tag-and-release, is the most straightforward to implement; simply tag some data of interest and observe where it goes in the system. However, the other two classes of policy significantly extend this model. By tagging a very large amount of data in the system, and by carefully managing the merging of those tags (at multioperand instructions for example), a large amount of information about the source of data can be determined. In our example system, we were able to trace data starting from an outgoing packet, back through the execution of Web application, to the source of that data in the set of incoming packets. While the merging of tags can be one of the trickier points of flow-source tagging, the final class of policies (confluence tagging) explicitly takes advantage of these merge points. Regions in the program where two or more dataflows are colliding are likely to be points of interest, and we have developed an example system which is able to map all of the interprocess and interfunction communication based on the identification of these collision points.

ACKNOWLEDGMENTS

The authors would like to thank Fred Chong, and the anonymous reviewers for providing useful feedback on this paper.

REFERENCES

AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. 2003. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM Press, 74–89.

BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. 2004. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI'04)*. USENIX Association.

BELLARD, F. 2005. QEMU, A fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*.

BITTON, D. AND TURBYFILL, C. 1988. A retrospective on the Wisconsin benchmark. In *Readings in Database Systems*, M. Stonebraker Ed., Morgan Kaufman, 280–299.

CASTRO, M., COSTA, M., AND HARRIS, T. 2006. Securing software by enforcing dataflow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (USENIX'06)*. USENIX Association.

CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. 2007. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*.

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (SSYM'04)*. USENIX Association, 22–22.

COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. 2005. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. ACM Press, 133–147.

CRANDALL, J. R. AND CHONG, F. T. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 221–232.

CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. 2005. On deriving unknown vulnerabilities from zeroday polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM Press, 235–248.

DALTON, M., KANNAN, H., AND KOZYRAKIS, C. 2007. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture.*

DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. 2004. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 292–302.

EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. 2005. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev. 39*, 5, 17–30.

ERLINGSSON, Ú., VALLEY, S., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. 2006. Xfi: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (USENIX'06).* USENIX Association.

GUPTA, D., YOCUM, K., MCNETT, M., SNOEREN, A. C., VAHDAT, A., AND VOELKER, G. M. 2006. To infinity and beyond: Time-warped network emulation. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation (NSDI'06)*. USENIX Association, Berkeley, CA, 7–7.

HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. 2007. Peerreview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*.

HAUSWIRTH, M., SWEENEY, P. F., DIWAN, A., AND HIND, M. 2004. Vertical profiling: understanding the behavior of object-oriented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM Press, 251–269.

HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. 2006. Practical taint-based protection using demand emulation. *SIGOPS Oper. Syst, Rev. 40*, 4, 29–41.

JOUKOV, N., TRAEGER, A., IYER, R., WRIGHT, C. P., AND ZADOK, E. 2006. Operating system profiling via latency analysis. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (USENIX'06)*. USENIX Association.

KICIMAN, E. AND LIVSHITS, B. 2007. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*.

KIM, H. C., KEROMYTIS, A. D., COVINGTON, M., AND SAHITA, R. 2009. Capturing information flow with concatenated dynamic taint analysis. In *Proceedings of the International Conference on Availability, Reliability and Security*. 355–362.

LARUS, J. R. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. ACM Press, 259–269.

LEVON, J. AND ELIE, P. Oprofile. oprofile.sourceforge.net.

NARAYANASAMY, S., POKAM, G., AND CALDER, B. 2005. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE, 284–295.

NEWSOME, J. AND SONG, D. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*.

PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. 2006. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev. 40*, 4, 15–27.

QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y. Y., AND WU, Y. 2006. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*.

SHAW, Z. A. Mongrel: mongrel.rubyforge.org.

SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 85–96.

SWEENEY, P. F., HAUSWIRTH, M., CAHOON, B., CHENG, P., DIWAN, A., GROVE, D., AND HIND, M. 2004. Using hardware performance moniters to understand the behavior of java applications. In *Proceedings of the USENIX 3rd Virtual Machine Research and Technology Symposium (VM'04)*. ACM Press.

TIWARI, M., AGRAWAL, B., MYSORE, S., VALAMEHR, J., AND SHERWOOD, T. 2008. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. IEEE Computer Society, 94–105.

TIWARI, M., WASSEL, H. M., MAZLOOM, B., MYSORE, S., CHONG, F. T., AND SHERWOOD, T. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, 109–120.

VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. 2004. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 243–254.

VENKATARAMANI, G., ROEMER, B., SOLIHIN, Y. AND PRVULOVIC, M. 2007. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*.

XU, M., BODIK, R., AND HILL, M. D. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM Press, 122–135.

XU, W., BHATKAR, S., AND SEKAR, R. 2006. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium (USENIX-SS'06)*. USENIX Association.

ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. 2006. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (USENIX'06)*. USENIX Association.

Received January 2010; revised July 2011; accepted August 2011