Understanding and Visualizing Full Systems with Data Flow Tomography

Shashidhar Mysore Bita Mazloom Banit Agrawal Timothy Sherwood

Department of Computer Science, University of California, Santa Barbara

{shashimc, betamaz, banit, sherwood}@cs.ucsb.edu

Abstract

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 Data Flow 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

Keywords Data Flow Tracking, Tomography Understanding

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 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 visualizing the full system data flow, 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, 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.

Data flow 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, 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:

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

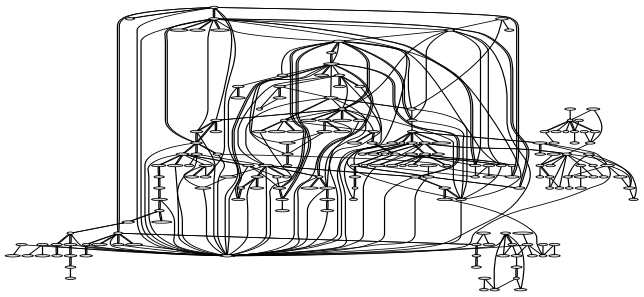

Figure 1. The function-level control flow graph of a very simple server program that awaits a socket connection and replies "Hello World". Even for such a simple program, there are 161 nodes (function invocations) and 409 unique edges (function to function call transfer). While control flow is an important 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.

- 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 data flow 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 data flow 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.

While we describe our prototype system in a fair amount of detail, it should be noted, before we go further, that this paper is completely qualitative (there are no bar graphs). 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 low level nature of our field 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 5) and in a discussion of the numerous system level concerns (in Section 4).

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). 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 (J. Newsome and D. 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 (Guru Venkataramani, Brandyn Roemer, Yan Solihin and Milos Prvulovic 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 (J. Newsome and D. Song 2005; Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou and Youfeng Wu 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 run-time guards to enforce integrity and access control policies (Úlfar Erlingsson et al. 2006; Castro 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 pro-

filing 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; 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 run-time 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* data flows, where specific pieces of data are used throughout the system, or how different components interact and communicate. Due to a necessarily heavy weight implementation, Data Flow 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. 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, Data Flow 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 (Kiciman and Livshits 2007; Chong et al. 2007; Haeberlen et al. 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. (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 Data Flow 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 data flow analysis to aid in understanding *full* systems: the processor, operating system, middleware, application, and network, all together.

3. Data Tagging for Tomography

Data flow 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. We simply treat that region of execution 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 data flow 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.

We have identified three different classes of data flow policy, uniquely suited to tomographic needs. To describe them more precisely, we introduce the following nomenclature.
Tag Insertion: the locations in the I/O, Memory, or Code where new tags are created.
Tag Propagation: the rules which govern, given operands of a specific tag, how an instruction should derive the tag of the data it produces.
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.

Each of the following classes of policy rely on the underlying mechanism of feed-forward data flow tracking with tags and differ in one or more of those three specific ways.

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 5. 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, mod-

ules, 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.

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.

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 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.

To understand the specifics of where we insert tags, how we propagate tags, and how we make sense of the gathered data, we need to first describe some of the details of our experimental system.

4. System-Level Issues

Having explained the different classes of data tagging policies required for tomography, in this section we discuss what that implies for a real system.

We implement our tagging policies on a networked set of virtual machines that are capable of propagating tag information both within the processor on every operation, and across the network as frames are exchanged. We maintain a tag map for every byte (as opposed to every word) of the physical memory. This tag map is treated as a general purpose identifier rather than a simple boolean or sets of independent booleans as in prior approaches. Four bytes of tag data per byte of physical memory is used, which gives us enough leeway 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). While the overhead of maintaining such a tag map is certainly very high both in terms of memory and computation, our goal is to both explore the potential of data flow tomography and to try and lose as little information as possible. In addition to the memory tags, every register in the processor model is extended with tags, and the virtual network card is modified to tunnel tags along with every Ethernet frame.

QEMU (Bellard 2005), an x86 virtual machine, provides the basic virtualization needed to implement and demonstrate our tomography tool. While we do not discuss performance further in this paper, with some trivial optimizations, this simple approach it is fast enough to do all of the data flow tracking within the machine and across the network, to log all the necessary information onto the Host disk, while still being capable of successfully hosting a web server (composed of Mongrel, Ruby, the Rails framework, Perl, and MySQL) serving requests at a reasonable rate with no timeouts.

As described is Section 3, the tag flow policies, and their implementation on a real system, can be thought of as three distinct pieces: tag insertion, tag propagation, and tag extraction. Tag insertion requires mechanisms by which an application/OS developer can associate tags in the underlying virtual machines with user-understandable data in the virtual system (packets or variables for instance). Tag propagation requires a mechanism to propagate tags within the system (through memory, registers, and potentially peripherals) and across systems in a networked environment. Tag extraction requires a way of mapping the hardware level tags, and the data flows 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.

While Section 5 concentrates on three different policies, in the rest of this section we describe how the above-mentioned requirements, which cross-cut all of the different policies, have been met on our prototype 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 (for example (*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 run-time.

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 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 data flow graph.

Network Tag Propagation: While Section 5 concentrates on different data flow propagation policies, propagation through the network uses the same infrastructure (because data is just transferred not transformed). The bottom of Figure 2 shows how the virtual machines communicate amongst themselves through a socket-based emulation of the Ethernet. The $eth1$ of the machine on the left running the Mongrel webserver communicates to the $eth0$ 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 $eth0$ of the host machine on the left, that frame gets routed to the tap interface, and the guest hardware pulls it off of its $eth0$ device. However, when the guest machine on the right wants to communicate to the external world, it writes a packet to its $eth0$ device. The socket based Ethernet emulation then reads this packet and sends it to the $eth1$ of the machine on the left (this channel is nothing but a TCP/IP socket). Once $eth1$ of the machine on the left receives this packet, it strips the packet of all the tags associated with this packet and then forwards it to the $eth0$ on the same machine, which is in turn read on the $tap0$ of the host machine on the left and forwarded to the external world through its $eth0$.

Tag Extraction: 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 the 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.

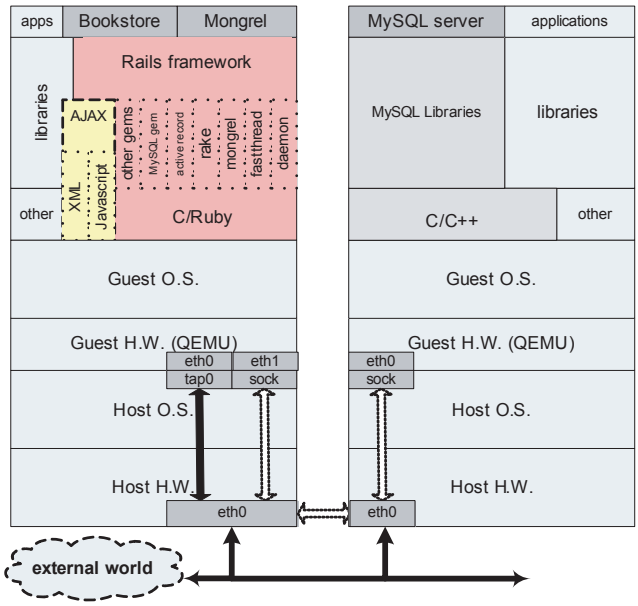

Figure 2. Our system setup includes the host hardware, a host OS, a guest hardware (QEMU), and a guest OS. The application, in this example an online Bookstore written in Ruby and running on the Mongrel Server, can communicate in a tag-preserving way through the network to other machines running various 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 ($eth1$ - for traffic within the virtual network) and the other does not ($eth0$ - so that normal external communication is possible)

4.1 System Setup

Figure 2 also gives an overview of the layers involved, and the modifications to the virtual machine required to build our Data Flow Tomography tool. The topmost layers show our example application layers which are unaware of the underlying data flow tracking. Here, Mongrel (Shaw) serves an online bookstore application while a MySQLD backend 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 webserver 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.

5. Example Tomography Scenarios

Using the infrastructure described in Section 4, we have built three example systems to help us map various aspects of the systems described in Section 4.1.

5.1 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 data flow 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 3 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

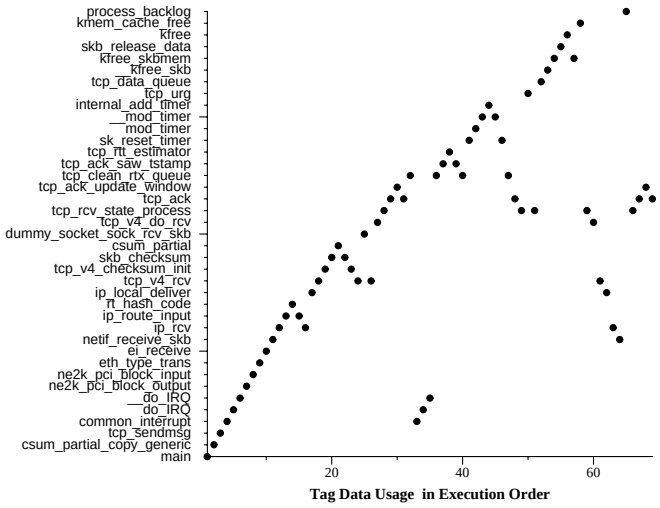

Figure 3. Execution of functions in the order of data flow. At step 0, the "Hello World" string is tagged by the server application (by communicating to the Tomography tool through the serial port). As the server executes, and as functions touch and spread the tagged data, a list of relevant procedures is created. The y-axis shows the procedures names of those functions touching the tagged data as the server executes.

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 3 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 even machines talking to one another. Later, in Figure 6, we will revisit this idea for a full web server (running AJAX, Ruby, Rails, etc.) and show that even tracking three different data flow 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). Data flow tomography allows us to slice through the otherwise unwieldy number of functions invoked during, even a single second, of web server operation.

5.2 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 Data Flow 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 pre-existing application-level knowledge). In fact, we show below that because we track dependencies at such a fine granularity, Data Flow 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 "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. In fact, we tag every byte of every incoming packet with a unique color (just an integer) so that we can identify which outgoing network bytes were dependent on which of the incoming bytes. We tag the bytes sequentially in the order that they arrive off of the virtual Ethernet card.

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 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 webpage and receives the webpage in response. In figure 4 (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 4, 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 5, 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, though a Tag-and-Release marking each of those fields. Figure 6 shows exactly that, with bytes corresponding to the header, cookie, and user data (a more detailed explanation can be found in the figure caption). 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 6) in a single pass.

5.3 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 inter-function, inter-module, inter-process, and inter-processor communication.

Tag Insertion: As explained in Section 4, 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 non-zero seed tag, which can come either from the network or the application.

Tag Propagation: To keep the advantage of data-flow tracking, we divide the tags into 0 and non-zero. 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 non-zero, then the non-zero tag is propagated. If a store instruction executes and is attempting store data that is tagged as non-zero, 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

[1] here we mean a physical color as drawn in a visualization, not to be confused with data-flow tag colors

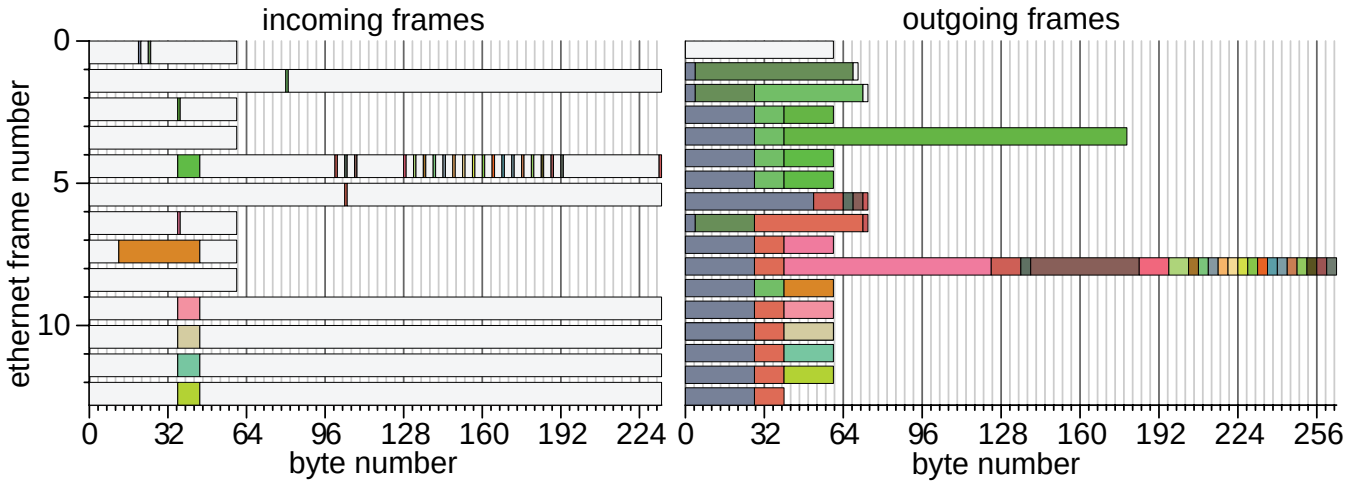

Figure 4. The figure on the left shows the incoming Ethernet frames with those regions that were discovered to flow from the input to the output for the simple wget experiment. The colors show the mapping between bytes in the incoming frames and they use in computing the data used in the outgoing frames.

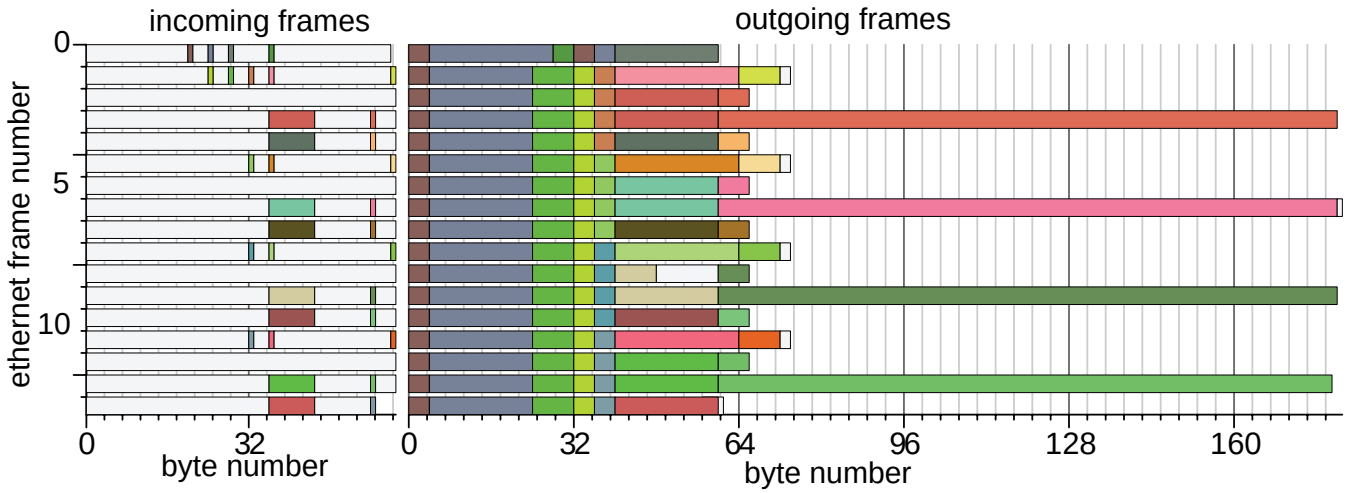

Figure 5. A simple web server running on Mongrel/Ruby shows the that there is a lot of correlation between the incoming and outgoing Ethernet frames. As in Figure 4, the colors of the outgoing frames are influenced by the colors of the incoming frames.

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 data flow as an example system, in addition to our simple "Hello World" server. Figure 7 shows how the tagged data flow in the simple server program. Each large rectangle represents a process and in this case the data flow 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 data flowed 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 data flowed 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 data flows across different processes in a scenario such as Apache and CGI/Perl is much more interesting and obviously more complex. Figure 8 is a representation of such a data flow as captured by our tomography tool. These visualizations are just one way of showing the data that can be gathered, and we are

[2] this is just based on the procedure name for now

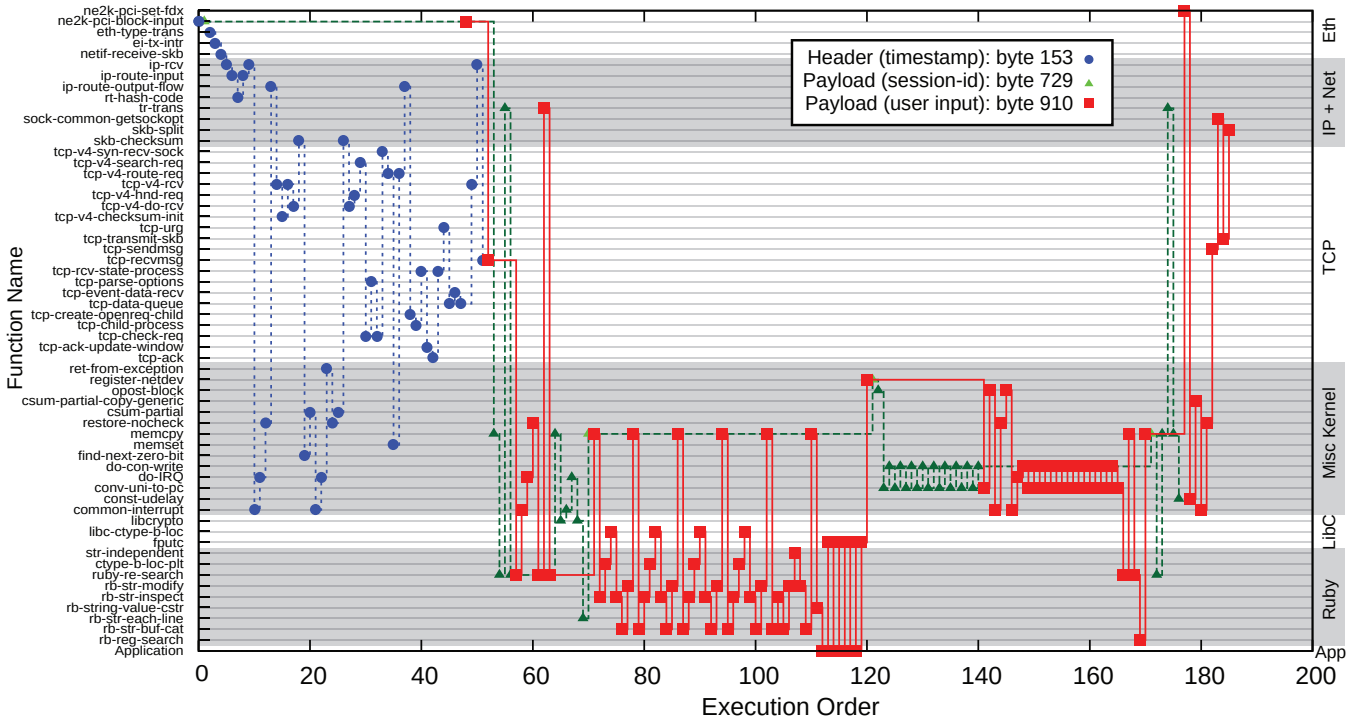

Figure 6. Execution of functions in the order of data flow for three different bytes of interest. The experiment shows a further step beyond the initial packet analysis from Figure 5. Once fields are identified through flow-source tracking, the semantics of those fields can be discovered by tag-and-release (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 Data Flow 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.

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.

6. Conclusions

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

First off, the method is inherently heavy weight compared to other approaches, both in 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 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. The scalability of such an approach as we increase the number of nodes beyond two is certainly a question.

A second challenge remaining is that, while memory and registers are tagged in our current implementation, the disk is not. Clearly, when we are talking about tagging a machine with multiple communicating processes, the physical memory seems to be the only place where tags can be safely kept (to ensure that memory mapped communication is handled correctly). Unless we tag the disk, as pages migrate to and from disk, their tags may be lost. In fact, any IO may result in "lost" tags (other than the network card and serial port as described in the paper). In our current implementation we ensure that no pages are ever swapped out to memory and we ignore through-file communication channels (which we manually verified were not used in these applications), but these are not acceptable solutions in the long term.

Third, a problem shared by this and most other ISA-level dataflow tracking approaches, is how data-flow 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 inter-process and inter-language communication) is needed.

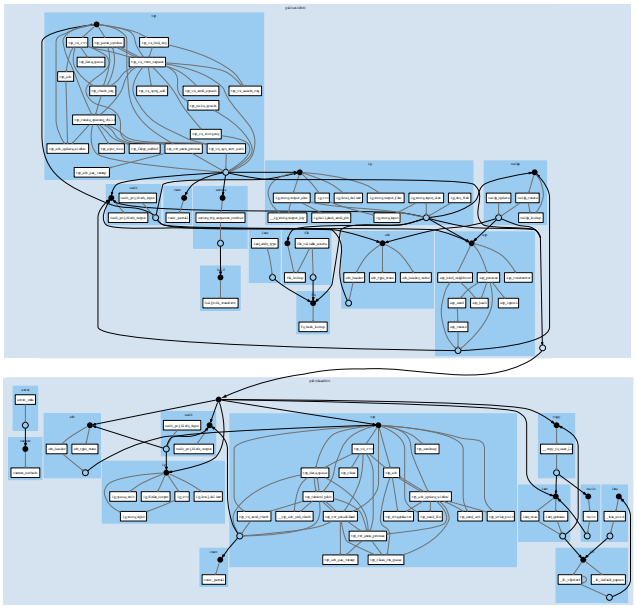

Figure 7. A Confluence Tagging based data flow diagram is shown for a simple network server program. The outermost rectangle represents one process, and the next inner rectangles represents modules within the process, and next smaller rectangles represent the individual functions which have exchanged tagged data.

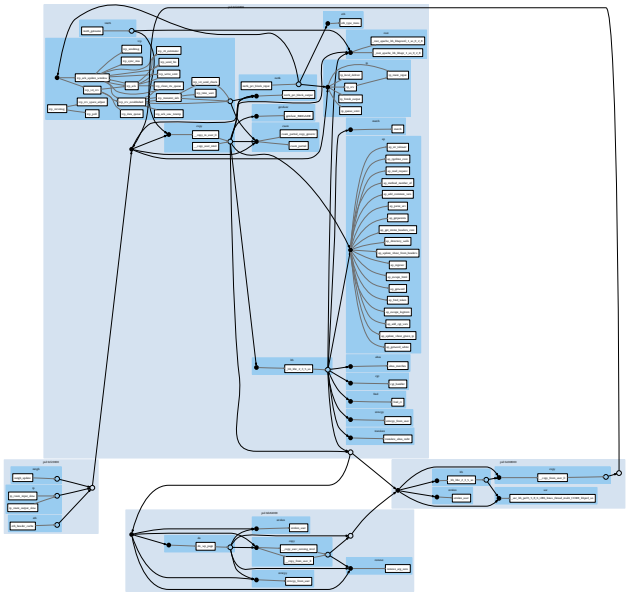

Figure 8. Dataflow diagram for apache web server handling CGI requests. The CGI request processing is handled as a separate process by Perl interpreter. The dataflow between multiple processes including apache web server, Perl, and kernel process is shown and each can be further zoomed in to understand the internal data flow within process or even within a module in the process.

Finally, due to the magnitude of data available from Data Flow 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 data flowing 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, Data Flow 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 multi-operand 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 data flows 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 inter-function communication based on the identification of these collision points.

Acknowledgments

The authors would like to thank Fred Chong, Mohit Tiwari, and the anonymous reviewers for providing useful feedback on this paper. This work was funded in part by NSF Career Grant CCF-0448654, CNS-0524771, CCF-0702798.

References

Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 74–89. ACM Press, 2003. ISBN 1-58113-757-5.

Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

F. Bellard. QEMU, A fast and portable dynamic translator. In *USENIX Annual Technical Conference*, April 2005.

Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *In Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In

SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium, pages 22–22. USENIX Association, 2004.

Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147. ACM Press, 2005. ISBN 1-59593-079-5.

Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6.

Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248. ACM Press, 2005. ISBN 1-59593-226-7.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, 2007.

Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Microarchitecture, 1997. Proceedings. Thirtieth Annual IEEE/ACM International Symposium on*, pages 292–302. IEEE Computer Society, 2004. ISBN 0-8186-7977-8.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39(5):17–30, 2005. ISSN 0163-5980.

Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks. In *Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

Guru Venkataramani, Brandyn Roemer, Yan Solihin and Milos Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, February 2007.

Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *In Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: understanding the behavior of object-priented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 251–269. ACM Press, 2004. ISBN 1-58113-831-9.

Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. *SIGOPS Oper. Syst. Rev.*, 40(4):29–41, 2006. ISSN 0163-5980.

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

Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.

Emre Kiciman and Benjamin Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *In Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

James R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269. ACM Press, 1999. ISBN 1-58113-094-5.

John Levon and Phillippe Elie. Oprofile: oprofile.sourceforge.net.

Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 284–295. IEEE Computer Society, 2005. ISBN 0-7695-2270-X.

Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, 2006. ISSN 0163-5980.

Zed A. Shaw. Mongrel: mongrel.rubyforge.org.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-804-0.

Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance moniters to understand the behavior of java applications. In *USENIX 3rd Virtual Machine Research and Technology Symposium (VM'04)*. ACM Press, 2004.

Úlfar Erlingsson, Silicon Valley, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: software guards for system address spaces. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.

Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004. ISBN 0-7695-2126-6.

Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 122–135. ACM Press, 2003. ISBN 0-7695-1945-8.

Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. USENIX Association, 2006.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.