Towards Practical Taint Tracking

Andrey Ermolinskiy
Sachin Katti
Scott Shenker
Lisa L Fowler
Murphy McCauley

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-92
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-92.html

June 5, 2010

Towards Practical Taint Tracking

Andrey Ermolinskiy Sachin Katti Scott Shenker Lisa Fowler Murphy McCauley

Abstract

This paper proposes several technical measures that significantly improve performance and largely limit kernel taint explosion in a XEN- and QEMU-based taint tracking system.

1 Introduction

Full-system, fine-grained taint tracking is a fundamental primitive that can be used for a number of purposes, most notably for tracking the flow of information through a system. Taint tracking is conceptually simple: memory containing data is tagged with a "taint", and the tag is tracked as contents of the memory are computed upon and moved. There is a large and growing literature of various taint-tracking techniques [9, 1, 15, 5, 19], and there are many cases where taint-tracking has proven both practical and beneficial. However, if one wants to avoid rewriting the OS or applications (which is our focus in this paper), it has proven extremely difficult to implement *practical* full-system fine-grained taint tracking systems.

The typical implementation approach in this case is to instrument hardware emulators such as QEMU with tracking instructions. Hardware emulation is extremely slow, so, to improve performance, modern implementations [5] use emulation only for those regions of code that interact with tagged data. These efforts have proved quite useful for binary analysis [9] but the performance is still not adequate for real-time use. In addition, fine-grained taint tracking often results in control information such as kernel data structures being tagged accidentally, which then amplifies into a *taint explosion* that leaves significant portions of the data unnecessarily tainted.

These two concerns, poor performance and taint explosion, have rendered full-system fine-grained taint-tracking impractical for deployment outside of a laboratory environment. This paper presents our progress in addressing these two problems and thereby making fine-grained taint-tracking more practical.

Our approach (which we call PTT for *practical taint-tracking*) begins with completely standard building blocks: PTT uses the Xen hypervisor to run the tracked system within a virtual machine, and (as in [5]) dynamically switches execution on to and off of the QEMU hardware emulator as tracking is required. To improve performance, we track tags at a higher abstraction level and in an asynchronous fashion. Specifically, instead of instrumenting the micro-instructions QEMU generates to track data, we create a separate stream of tag tracking instructions from the x86 instruction stream itself. This provides two benefits:

- For the tracking instructions, we avoid the amplification that occurs when we go from one x86 native instruction to multiple QEMU micro-instructions.
- Since this is a separate stream, the tracking can be done asynchronously without stopping the native emulation.

In essence, this approach is both directly more efficient (because it generates fewer tracking instructions), and it also allows us to execute those instructions asynchronously; it is the combination of these two effects that give PTT better performance.

However, as we explain later, it was difficult for us to provide direct comparisons with previous work. In the two specific cases where we could do so, PTT achieved a slowdown of roughly 1.4× (compared to native Linux execution), as opposed to previous efforts (in [19], based on [5]) in which the two comparison cases had slowdowns of roughly one and two orders of magnitude, respectively. We hasten to note that these two comparison cases were very optimistic ones, and there are other cases where our slowdown is much higher (roughly 20×). We discuss our results in greater detail later in the paper.

For taint explosion, we show that the source is accidental tainting of kernel data structures via a few system calls. By interposing at these specific system calls, and securely scrubbing taint, we prevent accidental tainting of kernel data structures. This prevents incorrect taint propagation to other data, and effectively eliminates taint explosion in practice.

The techniques we used to achieve these performance advances are not, by themselves, particularly deep or novel. However, the cumulative effect is to make taint-tracking significantly more practical. In fact, PTT is now usable, complete with a functioning GUI, and we have edited portions of this paper with it!

Before proceeding, we note that there is another problem with taint-tracking: Implicit information flows (such as copying by branching) do not leave behind a taint trail. However, here we are focusing on making taint tracking practical, not augmenting taint-tracking with other information flow tracking properties. Thus, while implicit information flows are an important issue for some applications of taint-tracking, we do not address this problem here.

This paper starts with a brief overview of our system (Section 2), then describes our design in greater detail (Section 3). We then examine the PTT's performance (Section 4) and taint-propagation (Section 5) properties. We end with a short discussion of related work (Section 6) and a brief conclusion (Section 7).

2 Design Overview

This section provides a broad overview of the design of our data tracking prototype. To provide context for the material below, we first quickly summarize PTT's overall architecture. Applications run in a protected virtual machine on top of a hypervisor. If the application accesses tainted data, the hypervisor switches the virtual machine from native virtual execution to an emulated taint-tracking processor running as a user-space application in a control VM. The application then executes in the emulated context, while the taint tracking processor produces a second stream of instructions whose sole purpose is to track taint. Once the application ceases to manipulate tainted data, the hypervisor can revert the system back to native virtualized execution.

We now describe our taint model, and then our taint tracking mechanisms.

2.1 Taint Model

Taints are attached to pieces of data, at its lowest granularity they can be attached to every byte of data, though in typical usage we expect large contiguous regions to have the same taint. PTT intentionally leaves the semantics of the taints opaque (PTT supports general 32-bit tags rather than single taint bit, and these tags are sometimes referred to as *colored taints* or *tints*). Systems can flexibly use taints to track whatever they wish. For example, a typical usage is to taint data coming in on an untrusted connection, and prevent it from being executed as code. Similarly, administrators can flag sensitive data with a taint describing the nature of its sensitivity, and block its exfiltration if appropriate.

Following [10], at a conceptual level the behavior of our system is dictated by three policies:

- **Input Policy**: The input policy determines what data is tainted and what taint to apply. For example, an input policy might be to taint all data from a socket with a particular label, and track it as it flows through the system. Of course, users and administrators can always mark taint on files and data themselves.
- **Propagation Policy**: A propagation policy determines how taint is propagated from one location to another. By default, our propagation policy carries taints from data sources to data sinks for all data movement (`MOV, MOVS, PUSH, POP, ADD, CMOV`, *etc.*) instructions.
- **Assert Policy**: An assert policy determines when to take action and what actions to take, based on the taint. For example, for data exfiltration, the appropriate assert policy would be to check taints and prevent exfiltration only at exit points, such as writing to a network device. By default, we worked with an exfiltration-style assert policy.

An alternative taint model is one used by designs such as Histar [17]—that is to attach taints to higher-level objects such as files, processes, threads, sockets, *etc.* While perfectly suited for the contexts in which these designs were proposed, this higher-level approach fails our requirements in two ways: To track at this higher semantic level requires rewriting the OS and/or applications, and it cannot track information at granularities smaller than these high-level objects (*i.e.*, it cannot distinguish between the confidential and non-confidential portions of a file). However, these approaches do track data through implicit information flows, while our approach does not.

2.2 Taint Tracking

Our goal in designing PTT was to build a comprehensive, deployable, and usable taint tracking substrate. Specifically,

- To be *comprehensive*, a taint tracking substrate should track taint persistently across storage, compute, and communication channels. Hence, we need to persistently track tainted data in the filesystem while it is being computed upon by applications and when it is either externalized or received on the network.
- To be *widely adopted*, a taint tracking substrate should work with existing applications and operating systems. While there are other approaches which can build simpler information flow tracking systems by modifying OSes and applications, their use in deployed systems is improbable, given the large investment already made in existing systems.
- To be *adopted in everyday use*, a taint tracking substrate has to be efficient and not demonstrably impair user-perceived performance. Second, it has to be parsimonious in how it propagates taint, yet maintain correctness. Prior work on taint tracking has been plagued by taint explosion, *i.e.*, accidental tainting of data that was not supposed to be tainted.

Figure 1 sketches a high-level picture of our system architecture. The key components are a hypervisor to isolate and securely interpose between the operating system and the hardware, an emulator to emulate and track instructions that operate on tainted data, a taint-aware file system that persistently stores taint information, and a network stack that adds taint metadata to packets. We describe each component below by tracking the lifetime of tainted data in a file that a user or security application

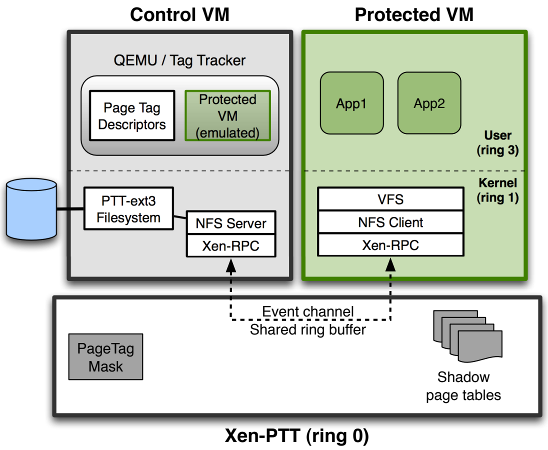


Figure 1—PTT System Architecture. The protected VM is emulated via QEMU in the control VM when it accesses tainted data. The PTT-ext3 filesystem is located in the control domain and the protected VM communicates with it via a shared memory RPC mechanism.

wishes to track. Note that this is only one application—tainted data could come from any source depending on the security application implemented (*e.g.*, to prevent possibly malicious code from being executed, one could track all data that arrives from an external untrusted network source).

The system being tracked is called the "protected" virtual machine (VM), and sits on top of the hypervisor. The control VM is where the emulator and the taint-aware filesystem resides. For simplicity in this example, we assume that the user taints an entire file with a single taint. The tainted file is stored in a taint-aware filesystem stored in the control VM. The filesystem maintains special metadata that keeps taint information for each block of the file. The filesystem serves as the main filesystem for the protected VM, and is accessed via a remote mechanism like NFS.

When an application in the protected VM opens the tainted file, the call is routed via the hypervisor to the taint-aware filesystem in the control VM. Before returning the file handle, the filesystem makes a call to the hypervisor, informing it of the file's taint information. The hypervisor also marks as dirty the pages in memory where the file contents are stored.

Shortly later, the protected VM tries to access the file data, and since the pages have been marked dirty, this causes a fault which is delivered to the hypervisor. The hypervisor saves the protected VM's context and switches to emulated execution in a user space process within the control VM. The emulator has direct mapped access to the VM's memory and thus continues to execute the machine in place.

The emulator is instrumented to track the taint of data being computed upon. Thus, as the emulator proceeds as normal to emulate the native x86 instruction sequence, it also executes parallel logic to update data structures that keep track of taint of data in the system. When the native instructions are no longer dealing with tainted data, the hypervisor switches the protected VM to native virtualized execution.

The final step is when the assert policy is invoked. For example, the policy might specify that an alert should be raised if tainted data is being externalized, namely, when the protected VM tries to write to a block output device such as a network card or a USB stick. The exception thrown could be an existing processor fault, such as an invalid opcode, or we can extend the processor with a specific "tainted" exception. The exception handler gives us an opportunity to write custom policy filters, which can either deny the write, modify the content being written, or just log it for audit.

Conceptually, the above sequence is the same as earlier work on taint tracking [5]. It provides comprehensive taint tracking end-to-end for existing systems, but fails to meet the important requirement of high performance and correct yet parsimonious taint propagation due to the following reasons:

- Taint tracking by plain instrumented emulation is extremely expensive. For example, even a simple computation on tainted data can incur a slowdown on the order of 95× when only 1/64th of the input file is tainted [19]. Such a slowdown is unacceptable in practice, and is unlikely to be deployed in everyday use.
- To be comprehensive, taint tracking has to track information flow across pointer references, *i.e.*, taint the referenced data with the same taint as the pointer. However, prior work [12] has shown that this leads to accidental tainting of kernel data structures. Soon, any other application interacting with the kernel also has its data tainted, eventually propagating taint to all data in the system. Such taint explosion renders the whole system ineffective (since the taint ceases to have the correct significance) and significantly impairs the performance of the system, as running in emulated mode suffers great performance penalty.

We design three novel techniques to solve the above two challenges in PTT. Specifically,

- PTT performs taint tracking in the emulator at a higher abstraction level than prior work. Emulators such as QEMU break down each emulated guest instruction into a series of micro-intructions. Prior work performs taint tracking by changing each micro-instruction to propagate taint, which immediately incurs a significant performance hit. PTT instead tracks taint directly at the native instruction level by producing a parallel stream of taint-tracking pseudo instructions that are executed separately (these are a small virtual extension to the x86 instruction set). The parallel stream size is of the same order as the native instruction sequence, and hence does not incur a significant performance hit.

- Second, PTT performs taint-tracking asynchronously. The key insight is that updated taint information is needed only at the point where assert policies are invoked. Hence, instead of synchronously tracking taint after the emulation of each native instruction, PTT takes the parallel stream of taint tracking pseudo-instructions and executes them asynchronously on another CPU core, letting the emulation proceed at full speed and returning it to native virtualized execution as soon as possible. In Section 4 we show that asynchronous tracking performed at a level of abstraction that directly matches the architecture of the emulated machine produces a 50× performance improvement over prior work.
- Finally, we identify via empirical evaluation that accidental tainting of kernel data structures happens via a very narrow interface, typically via a few specific functions in the kernel. We design techniques to interpose such channels of taint explosion, and securely control taint flow such that kernel data structures do not unnecessarily get tainted. We propose several minor modifications to the Linux kernel that eliminate accidental tainting and effectively solve the taint explosion problem.

3 Detailed Design and Implementation

PTT uses the Xen hypervisor as its virtualization technology, and QEMU as the emulation engine to track computation on tainted data. However, these systems must be modified significantly in order to provide efficient taint tracking. We start our discussion by reviewing QEMU.

3.1 Overview of full-system emulation using QEMU

QEMU provides a full-system emulation environment for several widely-used CPU architectures. The emulated ("guest") machine executes within the context of a single user-space process on the host operating system; all of the components of the guest hardware state (its CPU registers and physical memory) are represented with corresponding data structures in the address space of the emulator process.

Like other powerful emulators, QEMU implements dynamic code translation mechanisms to achieve reasonably efficient emulation. This code translation is essentially a two-stage process. In the first stage, the "front-end" component disassembles the guest machine code one instruction at a time. Each guest instruction is decomposed into a series of RISC-like abstract micro-instructions. In the second stage, a dynamic code compiler translates sequences of these abstract micro-instructions into blocks of native code for the host architecture. The results of code translation are cached in a pre-allocated memory buffer, which allows QEMU to amortize the significant overhead of code recompilation.

To support memory accesses from the guest machine, QEMU implements a software-based version of the memory management unit for the guest architecture. This software MMU provides guest-virtual to guest-physical address translation. In addition, a software-based TLB provides highly-efficient mapping between guest virtual addresses and the corresponding addresses in the emulator's own virtual address space.

QEMU implements some fairly sophisticated optimization techniques. However, the overhead imposed by basic emulation (without taint tracking) is quite significant; as we discuss in Section 4 the slowdown can be as much as an order of magnitude relative to native code in our measurements with QEMU v0.10.0.

3.2 Extending QEMU with instruction-level taint tracking

To track tainted data in emulated execution, we have to instrument the emulated instruction sequence with new instructions to update taint. One possible method that prior work [5] employs is to augment the micro-instructions QEMU produces with extra logic to update taint data structures. This approach is straightforward and convenient. However, the price we pay is a severe performance hit.

To see why, consider the REP MOV x86 instruction, which applications can use to copy an arbitrarily-sized region of memory between a pair of virtually contiguous memory buffers. (The source buffer address is loaded into register `%esi`, the destination is loaded into `%edi`, and the count is loaded into `%ecx`). This instruction has a number of common uses. For instance, the Linux kernel uses this instruction to transfer file data from the page cache to a user-level buffer when servicing a *read()* system call. To emulate this seemingly simple instruction, QEMU converts it into a looped sequence of micro-instructions. Each iteration of the loop decrements the count by 1 and transfers one word of data using one load and one store micro-instruction. Extending the load and store micro-instruction with taint tracking logic means that memory taint labels are also updated one word at a time. The cost of traversing the memory taint data structures in each iteration of the loop can make this operation tremendously expensive (up to 2 orders of magnitude relative to the actual data transfer).

The key limitation is the low level at which taint tracking is performed. Instrumenting QEMU micro-instructions to taint track automatically implies that we also have to track taint through all the temporary registers defined by the QEMU micro-architecture. Though this tracks taint correctly, it also significantly amplifies the number of taint tracking operations needed. A second important consequence is that taint tracking has to be necessarily sequential; the taint data structure has to be updated *in situ* with the emulated instruction sequence. This introduces substantial user perceived overhead, since users have to wait for taint tracking to finish before their application returns.

PTT conceptually uses the same approach to taint tracking, yet does not experience the performance hit due to two fundamental operational innovations: First, PTT moves taint tracking to a higher abstraction level; and second, PTT leverages the higher abstraction level to perform parallelized asynchronous taint tracking. We explain both of them below.

3.2.1 Taint tracking at a higher abstraction

PTT tracks taint by interposing at the first stage of QEMU code translation and synthesizing a separate sequence of taint tracking instructions directly from the guest x86 instructions. In abstract terms, we define a new set of instructions for a **taint processor** (a virtual hardware extension to the x86 architecture). During code translation, we examine each guest x86 instruction and generate the corresponding taint tracking instruction(s). The taint tracking instructions are separately executed to update taint data structures. The main advantage is that we avoid the overhead of tracking taint through QEMU's internal data structures via this approach, resulting in significant performance improvements.

For example, in the REP MOV instruction scenario discussed above, the PTT translator emits its taint tracking equivalent: REP_SET(Dest=Mem, Src=Mem). When handling this instruction, the taint tracker carefully examines the buffer size and alignment properties and optimizes the transfer accordingly. In a common scenario, this instruction is called with page-sized memory buffers and the source page holds uniformly-tainted data. In this case, it suffices to transfer a single page-level taint value, instead of copying word-level taints one at a time. Assuming 4KB-sized memory pages and 4-byte words, this technique reduces the number of accesses to memory taint data structures by a factor of 1024. Note that this optimization is enabled by the presence of higher-level instruction semantics. In this example, they allow the taint tracker to recognize a transfer between contiguous regions of memory and handle it accordingly—something that would be difficult to do once REP MOV has been decomposed into a loop of micro-instructions that performs word-level transfers.

Granted, our approach has costs. First, the implementation is more complex than prior approaches. We have to deal with the rich semantics of the x86 instruction set while generating the taint tracking instructions (*e.g.*, CMOV can mean either copy or no-op depending on the run-time value of a condition code). The richness implies that a single new taint tracking instruction will be insufficient, however we find that the taint propagation effects can be efficiently represented using a small number (16) of well-chosen taint tracking instructions. Second, creating a special taint processor for the x86 instruction set also means that the extensions will only work for x86 architectures. To perform taint tracking on a different architecture (*e.g.*, PowerPC), we will have to design a new virtual instruction set and a new code translator. However, given the dominant deployment of x86 machines, we believe this loss of generality is a prudent cost to pay for the performance improvements.

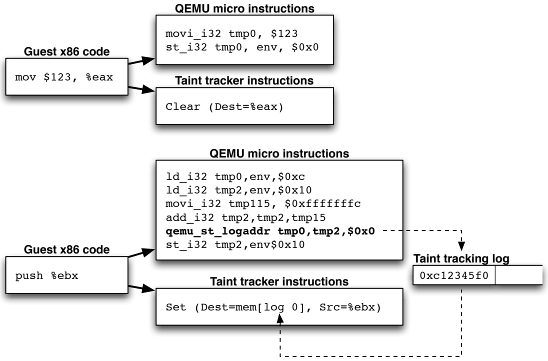
3.2.2 Parallelized asynchronous taint tracking

The second significant contribution of our system is the design and implementation of asynchronous parallelized taint tracking on multiple cores.

The key insight is that emulation and taint tracking are largely separable, and can be performed essentially in a parallel asynchronous fashion. Doing this operationally is made possible by our virtual taint processor: the taint tracking instructions emitted by the code translator are executed in parallel on a separate core in an asynchronous fashion. Decoupling emulation and taint tracking also orthogonalizes them. Advances made in emulation now directly benefit PTT and can be used with minimal changes.

The main complication with this basic scheme is that in some cases, the information needed to fully specify a taint tracking operation may not be available at the time of code translation. Consider `mov (%ebx), %ecx`, which loads a word of memory into register `%ecx` from the memory address specified in `%ebx`. While the destination operand is known at compile time, the taint source is a memory address given by the content of `%ebx`. How does the separately-executing taint tracking instruction sequence find this memory address? To supply the required information, we instrument the emulated instruction sequence with logic that temporarily stores the values of the operands in a memory-based log. To efficiently perform log updates, we implemented a new variant of the load micro-instruction (*qemu_ld_logaddr*), which appends the physical memory address of the source operand to the log. When the taint tracking instruction sequence is executing, it consumes operands of this log to correctly propagate taint.

In operational terms, the emulator and the taint tracker are in a producer-consumer relationship and use the log to coordinate their activities. The log resides in a static pre-allocated memory buffer and is divided into a number of smaller "log regions" (units of synchronization). After executing a basic block of emulated code, the emulator thread (producer) synthesizes the corresponding "taint block descriptor" and appends it to its current region. This data structure holds a pointer to the block of taint tracking instructions (generated by the code translator), followed by the list of arguments to these instructions that were resolved at runtime. The emulator thread can



```
mov $123, %eax
```

```
movi_i32 tmp0, $123
st_i32 tmp0, env, $0x0
```

```
Clear (Dest=%eax)
```

```
push %ebx
```

```
ld_i32 tmp0,env,$0xc
ld_i32 tmp2,env,$0x10
movi_i32 tmp115, $0xfffffffc
add_i32 tmp2,tmp2,tmp15
qemu_st_logaddr tmp0,tmp2,$0x0
st_i32 tmp2,env$0x10
```

```
Set (Dest=mem[log 0], Src=%ebx)
```

```
0xc12345f0
```

Figure 2—Example of QEMU code translation for emulation and taint tracking.

now schedule the taint tracking thread (consumer) on a separate core while it moves on to emulate the next basic block of guest instructions.

Figure 2 illustrates the code generation and logging mechanisms with a simple example. The first guest instruction loads a constant value into register `%eax` and the corresponding taint tracking instruction simply clears out the register taint. The second guest instruction pushes the value of `%ebx` onto the stack. For this instruction, we must propagate the current register taint from `%ebx` into a 4-byte memory word whose starting address corresponds to the top of the stack (as given by `%esp`). Of course, the exact address is not known at the time of code generation and to obtain this value, we use our new version of the QEMU *store* micro-instruction, which additionally writes the physical address of the memory location to the taint tracking log. In this example, the *qemu_st_logaddr* instruction is generated to append the physical address of the stack pointer (whose virtual address is held in `tmp2`) to the log. Accordingly, the taint tracking instruction encodes `%ebx` as the source operand and the destination operand references the log position, to which the stack pointer address will be written at runtime. (This log position is specified relative to the start of the current code translation block.)

The log size is a crucial parameter. Taint tracking is more expensive than emulation. Thus, if the log size is small and the emulator (producer) runs out of space in the log, the emulator must block until the taint tracker (consumer) catches up and clears up space. At such points, we effectively return to lock-step synchronous taint tracking. While this scenario is theoretically possible, we expect that it will rarely occur in practice. The reason is that in most forms of interactive computing involving humans and there are always gaps in computation (*e.g.*, the user pausing before entering more text). These gaps can be gainfully exploited by the taint tracker to update taint and empty the log. We expect these gaps to be sufficiently large relative to the time taken to perform taint tracking, thus allowing us to sidestep the full log scenario.

However, note that in certain scenarios we have to explicitly force synchronization between emulation and taint tracking. Specifically, at any point where an assert policy has to be enforced, or the guest system is returning to native virtualized execution from emulated mode, taint tracking data structures have to be fully updated. The reason is that both scenarios depend on the correctness of taint propagation to proceed further. In this case, we explicitly suspend the producer (emulator), and wait for the taint tracker (consumer) to complete.

3.2.3 *Returning from emulated execution*

A final challenge is determining in which situations it is safe, beneficial, or necessary to exit emulated execution and resume native execution of the suspended Xen guest domain. Conceptually, one can exit emulated execution as soon as all the CPU registers are untainted. However, this might not be prudent, since the very next instruction might again access tainted data necessitating a switch back to emulation. Frequent context switches between emulated and virtualized execution will be expensive and contribute to significant slowdown.

PTT handles this problem by introducing some delay before switching. Specifically, it keeps a counter of the number of consecutive native instructions in emulation mode that did not access tainted data. If the counter is above a certain threshold, it decides it is safe to exit emulated mode and return to virtualized execution. The threshold is empirically determined and is currently set to 50 instructions that do not access tainted data, as in [5] and [19].

Finally, it becomes necessary to exit the emulated mode to handle faults and software interrupts (including hypercalls) that result in a jump to hypervisor-level code residing in the highest privilege level (ring-0). The current architecture of Xen makes it difficult to emulate the execution of hypervisor-level code in a user-space process. Instead, QEMU temporarily suspends emulated execution, transfers the guest CPU context back to the hypervisor, and instructs it to perform a *temporary switch* to the native mode at the instruction that causes transition to the hypervisor. (*e.g.*, *int*0*x*82 for hypercalls). Temporary native execution terminates upon the return from the hypervisor context (*i.e.*, a hypercall handler or a fault handler). Immediately before transferring control to the guest domain (typically via the `iret` instruction), the hypervisor suspends the guest, transfers its CPU context back to the emulator, and instructs it to resume emulated execution.

3.3 Taint Storage and Trapping

PTT tracks taints persistently end-to-end. To accomplish this, PTT has to design specialized data structures that store taint in the filesystem and memory, and trap to the emulator when accessed. Below we discuss these components.

Taint Storage in Filesystem and Memory: PTT uses an augmented ext3 filesystem to store taints persistently. Taints are stored in each file's metadata on a block-by-block basis. PTT uses a two-level data structure to exploit spatial locality and store taints efficiently. At the first level, we maintain a 64-bit field called the *BlockTaintSummary* for each block of the file. If the entire block of data has a single taint associated with it, then the taint is directly stored in the *BlockTaintSummary*. Otherwise, it acts as a pointer to another larger data structure of variable length (the *BlockTaintDescriptor*), which stores the taints for each byte in the block. The *BlockTaintSummary* data structure is maintained in the metadata of the file, while the larger *BlockTaintDescriptor* is kept in a separate block. The *BlockTaintDescriptor* uses run-length encoding (RLE) to exploit the expected spatial locality in byte taints and reduce storage overhead.

A similar two-level data structure is used for taint storage in memory. The corresponding components are called *PageTaintSummary* and *PageTaintDescriptor*.

Trapping Sensitive Data Access An implicit assumption in our earlier description of taint tracking was that we would automatically switch to emulated execution as soon as tainted data was accessed. PTT uses a well-known technique based on *shadow page tables* to trap such accesses efficiently and detailed descriptions of this mechanism can be found in previous studies [19, 5].

4 Evaluation

In this section, we evaluate the performance overhead of our prototype implementation under a variety of workloads, which include synthetic microbenchmarks and real-world applications. We find the following:

- For a system with 10% of the data tainted, the computational overhead over native execution is 40%. In the worst case (when all data is tainted), the overhead is 20×. Both of these results appear to be significantly better than prior work, though we can only make exact comparisons in two specific senarios.
- The performance improvement is the result of a combination of high-level taint flow instrumentation and asynchronous parallel execution of taint tracking.
- PTT can support graphical windowing environments, ensuring a reasonable level of interactivity and application-level performance. To the best of our knowledge, PTT is the first online taint tracking system to demonstrate support for interactive workloads in a graphical desktop environment.

4.1 Setup

Our test machine is a Dell Optiplex 755 with a quad-core Intel 2.4Ghz CPU and 4GB of RAM. The hypervisor-level component of our implementation is based on Xen 3.3.0. The emulator and taint tracking modules (based on QEMU v0.10.0) run in the privileged Xen domain as a multi-threaded user-level process. The guest domain is configured with 512MB of RAM and one VCPU (as our current implementation does not yet offer support for multi-processor guest environments). The guest runs a paravirtualized Linux kernel v2.6.18-8.

Our experiments evaluate the overhead in the following configurations:

- **NL**: Linux on native hardware
- **PVL**: Paravirtualized Linux on Xen hypervisor
- **Emul**: Linux in a fully-emulated environment using unmodified QEMU
- **PTT**: Our prototype implementation of Practical Taint Tracking
  - PTT-S: Synchronous taint-tracking
  - PTT-A(*x*): Asynchronous parallel taint tracking with *x* MB of memory reserved for the log. We explore using log sizes of 512MB, 1GB, and infinite (∞)

We compare our system primarily with Neon [19] when possible. Neon builds on top of the on-demand emulation-based taint tracking system developed by Ho et al. [5], and hence provides a comparison to both systems most closely related to us. However, we were unable to get the code for [5] working on our system since it is based on a heavily outdated version of QEMU and has fragile dependencies, and as such would not work on our test systems. Thus, the only way we could provide direct comparisons was to run the same experiments as reported on in [19] and use their published results to compare the two systems.

4.2 Application-level overhead (data processing tools)

In this section, we evaluate the overall performance penalty of taint tracking in common usage scenarios, as perceived by potential users of the system. The goal is to measure if our taint tracking substrate can be used for everyday computing activities.

4.2.1 Copying and Compressing

We begin by considering two simple but very common data manipulation activities:

LocalCopy - Copying a partially-tainted file to another file in the guest filesystem using the `cp` command.

Compress - Compressing a partially-tainted input file using the`gzip` command. Compression represents a somewhat more stressful scenario as it involves a non-trivial amount of computation on the input data.

This choice of benchmarks also enables us to compare our results with that in [19]. While the results reported there focus on the case of sparsely-tainted input files, we also wish to understand the worst-case performance impact of taint tracking. To this end, we measure the performance with varying amounts of tainted data in the input file.

Similarly to [19], we use a 4-MB input file and measure the command completion time. Before the start of each measurement, we pre-stage the input file into the filesystem buffers on the host machine. This allows us to factor out the overhead of disk I/O (which remains constant across all configurations) and measure the fundamental overhead of taint tracking in the most stressful scenario (a CPU-bound task).

Figure 3 provides the performance results from this experiment, expressed in terms of the slowdown factor relative to *NL*. For lightly-tainted input files (10%), *PTT-A(512)* increases the running time by a factor of 5.3 for file copy and a factor of 5.8 for file compression. As the amount of tainted data grows, our system must spend more time in the emulated mode and the slowdown becomes much more noticeable. In the extreme case of a fully-tainted input file, our implementation incurs slowdowns of 15.9× and 21.1× for copying and compression, respectively. The copy operation involves no computation on tainted data—it merely transfers file data between user- and kernel-level memory buffers (typically using the *REP MOV* instruction). The page-level taint transfer optimizations introduced earlier allow the taint tracker to handle this scenario efficiently.

This experiment exercises the ability of PTT to transition efficiently between virtualized and emulated execution modes. Ideally, one would expect the slowdown to scale linearly with the fraction of taint, since the amount of taint should dictate the amount of time spent in emulated taint tracking mode. Our system does not show linear scaling because the heuristics for transitioning are not perfect. The heuristics err on the conservative side, keeping the system in emulated mode even if one could have transitioned back to native virtualized mode a bit earlier. Hence, the overhead at low levels of taint is larger than the linear scaling would suggest.

Finally, although we do not have enough data to draw definitive conclusions, we believe that these results yield a favorable comparison to Neon. In a scenario with $f = 1/64$, where f is the fraction of tainted data, Neon reports slowdown factors of 10× and 95× for file copy and compression, respectively. This is the only data point for this experiment reported in [19] (Table 3). As Table 1 below shows, taint tracking with PTT incurs significantly less overhead. The overhead for both file copy and compression is only 1.4× over native execution. In Figure 3, we can see the performance of PTT over the range of f, but have no corresponding data from [19] except for $f = 1/64$.

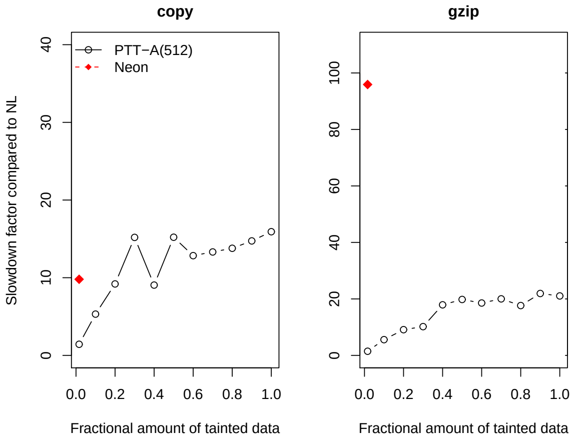


Figure 3—Performance overhead in the *LocalCopy* and *Compress* experiments.

	NL	PTT-S	PTT-A(512)
LocalCopy	0.011	0.0164	0.0158
Compress	0.062	0.0965	0.0928

Table 1—Command latency measurements (in seconds) for *LocalCopy* and *Compress* with $f = 1/64$

4.2.2 Searching

In the next experiment, we consider another common usage scenario: text search. We use the *grep* command to search the input data set for a single-word string and measure the overall running time. Our input dataset for this experiment is a 100-MB text corpus spread across 100 equal-sized files. These files reside on disk at the start of the experiment. We measure the command completion time in all configurations of interest and compute the slowdown relative to native execution, *i.e.*, *NL*. For PTT, we repeat the experiment multiple times, varying the number of files f marked as tainted. Unfortunately we cannot compare with Neon, since they do not report any measurements for this scenario.

Table 2 reports the results of this experiment for the most stressful scenario, *i.e.*, when all files are tainted ($f = 1$). As expected, the performance is significantly worse compared to native execution for this worst case scenario. PTT with asynchronous parallelized taint tracking is slowed down by a factor of 10×. For more modest amounts of taint, when only 10% of the files are tainted, the overhead is 46% over native execution.

Table 2 also quantifies the gain over synchronous taint tracking obtained by performing taint tracking in an asynchronous parallelized fashion. Asynchronous tracking us-

	NL	PVL	Emul	PTT-S Synch	PTT-A(512) Asynch
Completion time (s)	2.42	2.87	18.45	58.865	25.57
Slowdown factor	1.00	1.18	7.62	24.32	10.56

Table 2—Text search performance in the worst-case scenario ($f = 1.0$).

ing multiple cores roughly provides a 2.5× gain over synchronous sequential tracking on a single core. The best case gain that can be obtained from parallelization of taint tracking is if it creates an illusion that there is no taint tracking at all, *i.e.*, it has the same performance as pure emulation. PTT with parallelized tracking incurs a 40% overhead over pure emulation. This nontrivial overhead is to be expected, especially since the emulated instruction sequence has to be instrumented to write to the log operands that are needed by the taint tracking instruction sequence.

4.3 Benefits and limitations of asynchrony

Asynchronous parallelized taint tracking provides significant benefits as noted in the previous section. However, it comes with a small cost, specifically the log, which has to be updated by the emulated instruction sequence. The size of the log has a major impact on the performance of asynchronous tracking: a small log could get frequently filled up and force the producer (the emulator) to wait for the consumer (the taint tracker) to catch up. In this section, we evaluate the impact of log size on PTT's performance.

We focus on the absolute worst-case scenario—a CPU-bound workload on a fully-tainted dataset. We consider two such workloads: Compressing a file (using *gzip*) and sorting an array of integers (using the *qsort* library routine). We measure the running time for varying input sizes and repeat the measurements for three different versions of PTT with different log sizes:

1. *PTT-A(512)* with a 512MB log.
2. *PTT-A(1024)* with a 1024MB log.
3. *PTT-A(∞)* is a special case which is supposed to simulate the scenario where log size is not a concern. In this configuration, the producer never stalls due to lack of log space and it represents the absolute upper bound the on the degree of performance improvement attainable by optimizing the taint tracker. Viewed from a different angle, the gap between *Emul* and *PTT-A(∞)* quantifies the overhead of logging on the producer side.

Figures 4 and 5 report the slowdown relative to *NL* for all configurations of interest. *PVL* shows the baseline performance on Xen and *Emul* isolates the impact of emulation. As we can see, both compression and sort suffer an 11× slowdown in emulation. For a 1MB input file, compression runs 22.4× slower on *PTT-A(1GB)* and 22.5× slower on *PTT-A(512)*. As we expect, these numbers start to diverge as input size increases. With a 20MB input file, *PTT-A(1GB)* still offers sufficient log space to absorb most of the overhead, and thus the computation proceeds at a rate close to the upper bound given by *PTT-A(∞)*. An input file of size 20MB appears to be the point of transition for *PTT-A(1GB)*, beyond which the producer starts stalling on log space. According to our estimates, this transition occurs around the 5MB input size for *PTT-A(512)*. We can see that for both configurations, there is an asymptotic penalty as we increase the input size, plateauing at a 35× slowdown. Viewed collectively, these results validate our intuition that *PTT-A* can take advantage of underutilized memory resources to alleviate the burden of taint tracking.

Again, there does not exist much data on how previously proposed taint tracking systems behave at this level of stress, which makes it difficult to provide a direct comparison. We hope that the following data points, which we were able to collect from the literature, could serve as a starting point for such a comparison:

- Neon [19] reports overheads ranging from 10× to 95× for CPU-bound workloads when $1/64^{th}$ of the input file is tainted.
- The initial implementation of taint tracking using on-demand emulation on Xen [5] reports a 155× slowdown for what appears to be a CPU-bound task operating continuously on tainted data.

PTT clearly outperforms both these prior approaches, for systems with much larger levels of taint. Even with 100% taint with a computationally bound workload, the slowdown is only a factor of 22×.

4.4 Performance in graphical environments

In our last experiment, we study the impact of taint tracking on user interactivity and application-level performance in a graphical user environment. To this end, we deploy a full-fledged GUI stack (X Window server and the GNOME desktop environment) on top of our taint tracking substrate.

Graphical environments present a non-trivial challenge for fine-grained taint tracking systems such as PTT. Although such environments rarely impose high computational demands, the key challenge is dealing with window rendering systems when tainted data is present on the screen. A simple screen refresh will involve moving tainted data around, and hence lead to constant oscillating switches between native and emulated execution.

A direct application of PTT to a graphical environment with tainted data on the screen leads to significant per-

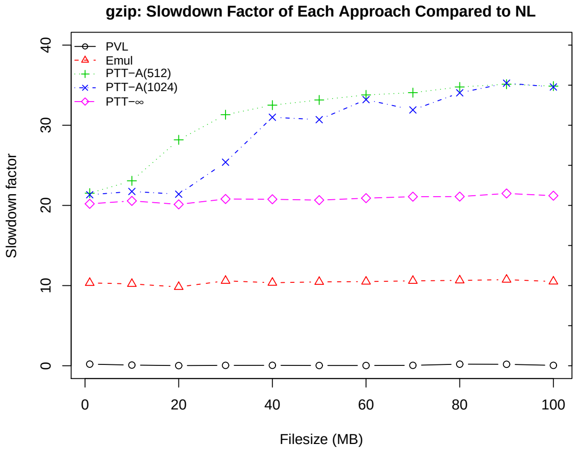


Figure 4—Application benchmark: `gzip` – Relative slowdown to *NL* for all configurations of interest. *PVL* shows the baseline performance on Xen and *Emul* isolates the impact of emulation. We can see that the worst-case performance is asymptotic at 35× slowdown, no matter the size of the log.

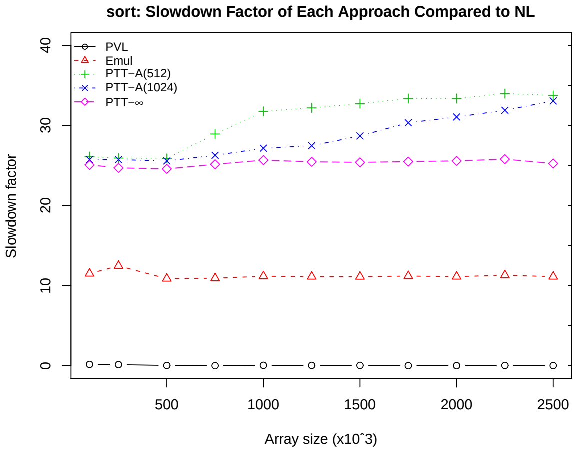


Figure 5—Application benchmark: `sort` – As with `gzip`, we can see that the worst-case performance is asymptotic at 35× slowdown relative to *NL*, no matter the size of the log. Unlike with `gzip`, there is more benefit to an increased log size.

formance impairment, to the point that the user perceives serious interactivity problems. The reason is a thrashing behavior resulting from repeated screen repaint operations. In typical usage scenarios, basic user actions (such as mouse movements and keystrokes) trigger application-level events that cause the window image to be recomputed. For instance, if a text window is showing tainted data and the user is entering text from the keyboard, each keystroke cause a page fault and transition to emulation. When the guest system finishes computing the new window contents (reflecting the keystroke) and relinquishes the CPU, we switch back to the native mode, but find ourselves re-entering emulation once again upon the next keystroke.

We found that the overall performance and usability of the system can be greatly improved in such situation by making a simple fix: *persistently* switching to emulation mode and remaining in emulation mode for as long as tainted data remains on screen. Keeping the system persistently in emulation also enables us to leverage significant benefits from asynchronous parallelized tracking. In fact, interactive graphical environments seem to be a fairly compelling use case for asynchronous taint tracking. Since the guest workload is interrupt-driven and proceeds mostly at human timescales, the taint tracker can easily keep up with the producer and the log helps absorb the short burst of computation resulting from basic user activity.

Persistent emulation with asynchronous taint tracking led us to a fully-operational and usable graphical environment. In this environment, users observe almost no perceivable degradation of interactivity for simple UI actions (*e.g.*, moving the mouse, entering text, scrolling).

To measure the performance impact on application-level operations, we instrumented the *gedit* text editor application and ran a simple user session. This session included launching the editor, opening a 1.2MB text file, making some changes, computing document statistics, and saving the file under a different name. Table 3 reports the results from this experiment and Figure 6 shows a trace of taint tracking log usage and CPU utilization in the control domain for this user session.

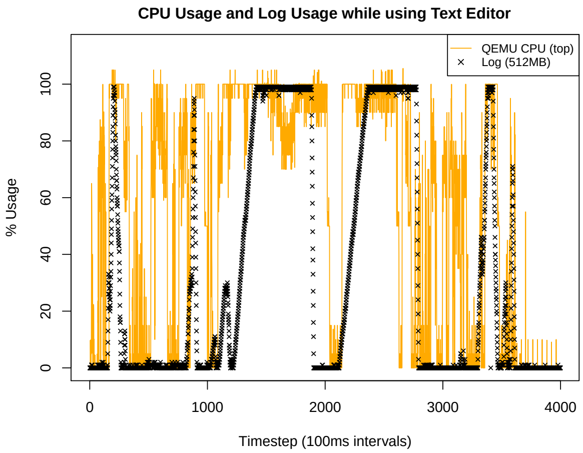


Figure 6—Time series of log usage and host CPU usage by the QEMU process. Note that the 100% CPU usage mark represents full utilization of both processor cores (producer and consumer).

Although the text editor was fully usable and responsive during this user session, the measured performance degradation was somewhat higher than we expected. Notably, the component of the overhead due to taint tracking does not increase from previous experiments in the text console environment. At the same time, the costs of basic system emulation increase to about 20×. Further investigation revealed the likely source of this discrepancy. The GNOME graphical environment on x86 makes extensive

Action	NL	Emul	PTT-A(512)
Launch editor	2403	7472	8741
Open file	302	2087	4634
Compute document stats	307	7309	12491
Save file	420	8300	15086

Table 3—Completion time (in *ms*) for a range of user actions in the *gedit* text editor experiment.

use of the SSE instruction set extensions and QEMU does not currently optimize the emulation of these processor features. Current versions of QEMU dynamically recompile arithmetic and memory access instructions into native code and try to optimize the use of host registers. At this time, QEMU does not perform JIT recompilation for SSE instructions and does not take advantage of the SSE registers available on the host processor. We expect that the overhead of our system will be reduced further with orthogonal improvements in emulation technology.

5 TAINT EXPLOSION

All prior taint tracking systems suffer from taint explosion, *i.e.*, the phenomenon where almost the entire memory content of the tracked system becomes tainted, even though only a small amount of tainted data is introduced into the system. Several papers [12, 15] have observed and measured this phenomenon and have questioned the usefulness of taint tracking given its profligacy in propagating taint accidentally.

Left unchecked, taint explosion significantly impairs the performance of our substrate in two important respects. First, it unnecessarily forces our system to spend more time emulating the guest VM and adds perceptible overhead. Second, it confuses users and security applications, since they have no way of telling if a piece of tainted data has been tainted due to the right reasons, or accidentally.

Upon further examination, we discovered that taint explosion starts primarily from one source: kernel data structures. Specifically we found from an empirical evaluation over a long-lived taint tracking trace, that in all execution paths that deposit taint into the kernel, only two code blocks in the Linux kernel are responsible for the initial transfer of taint from user space to kernel data structures.

In the first case, the transfer occurs when the system call entry routine writes the user registers to the kernel-level stack (some of them holding system call arguments), and the spread of taint begins when the system call handler routine accesses these arguments from the stack. In the second case, the transfer happens via the *copy_from_user* routine (and its variants). The kernel calls this function to transfer additional arbitrary-length system call arguments from an application-level memory buffer and this can cause taint to be transferred into kernel-level stack and heap areas.

Close examination of these system calls revealed that these taint propagation paths were completely accidental and were not actually propagating any information, *i.e.*, the user space data was not influencing the state of the kernel data structures. Subsequently, any other userspace process that interacted with the kernel would get tainted from these data structures and incorrectly propagate taint further in the control path. Soon, the entire system would become tainted in this fashion.

While the presence of these channels is not at all surprising, the interesting fact that emerged from our analysis is that no other channels exist. This finding motivates our solution to the problem of taint explosion: namely, identifying the system call channels through which accidental kernel tainting was occurring, securely intercepting them at the hypervisor level, and explicitly erasing the taint from system call arguments in the very first steps of system call handling for these channels. Our current implementation achieves this by issuing a hypercall, which the emulator intercepts.

We must, of course, ensure that any solution we adopt to address kernel taint poisoning does not interfere with the legitimate and explicit channels of information transfer the kernel is expected to provide. By examining the system call interface, we identified the following channels of explicit transfer:

- `sys_write` (and its variations): Taint transfer may occur because because the kernel temporarily stores application data in its filesystem buffers.
- `sys_send` (and its variations): Taint transfer may occur because the kernel temporarily stores application data in its socket buffers.
- `sys_ipc`: The kernel provides temporary storage for application data to support message-based IPC.

We do not intercept or scrub any taint for these channels.

With these modifications, we *eliminated* taint explosion from PTT for all practical purposes. Specifically, we ran a control experiment where a userspace application opens a tainted file, does some computation on it, and then closes it. Prior to the above solution, the whole guest VM would become fully tainted, even though no further computation occurred with tainted data. After the above solution is implemented, no further propagation of taint occurs after the tainted file is closed, while correct taint propagation (per policy) still occurs. We observe the same pattern in a number of other control experiments. While this may not cover all possible channels of taint explosion, we believe that our experiments do study the majority of practical usage scenarios and effectively eliminate taint explosion for these scenarios.

To see if these measures impaired performance, we measured the overhead of the taint scrubbing actions we perform for those specific system call channels. We use the

	null	stat	fork
PVL	0.248	0.952	242.21
PTT	0.557	1.593	243.00

Table 4—Latency (in μs) for several system calls in Paravirtualized Linux and PTT, as reported by LMBench.

LMBench benchmark to measure the latencies of three different system calls (*null*, *stat*, *fork*), and compare these numbers to the baseline native execution scenario of paravirtualized Linux (*PVL*). Table 4 reports the results of this experiment. We see that for a no-op (*null*) system call, the additional hypercall represents a significant (factor of 2.2×) penalty, but this overhead is much less noticeable when we consider non-trivial system calls.

Our taint scrubbing modifications to the guest kernel required modifying three source files in the Linux kernel source tree and adding 90 lines of new code.

6 Related Work

Our system builds on extensive prior work in information flow tracking (IFT) and dynamic taint analysis (or "taint tracking") in commodity operating systems and applications. Information flow tracking was introduced nearly 30 years ago as a technique for security policy enforcement [3, 4]. Early efforts focused on static analysis (such as the popular work in information flow control by Myers and Liskov [7]), with later techniques supporting dynamic information flow labels by combining static and run-time checking, such as [6] and [8]. Another recent work in dynamic information flow tracking [16] offers a language runtime that propagates policy objects along with data, and applies this policy by filtering objects at system I/O boundaries. However, each of these language-based techniques require complete application rewrites and the static techniques cannot support dynamic changes to policy without recompilation.

Process-level information flow control mechanisms [14, 17, 18] enable mandatory access control through the use of coarse-grained labels and operating system support. Leveraging the earlier works, the most recent effort [18] adds specialized hardware to gain great reductions in overhead and size of the trusted code base.

In order to perform information flow tracking in environments that do not provide such language, compiler, or OS support, recent efforts have relied on binary rewriting at run-time as a way of introducing IFT into commercial applications and operating systems. The most popular technique is that of "taint tracking," wherein program inputs are "tainted" and that taint is dynamically propagated along the control path of the program. As such, the technique of taint tracking has been used for addressing numerous security-related issues. For example, one body of work [9, 13, 15] applies fine-grained taint tracking mechanisms to the problem of malware detection and analysis. These approaches rely on full-system emulation and impose significant slowdowns, but performance overhead is a secondary concern for offline analysis. Other systems use taint tracking mechanisms to understand the issues of data lifetime and leakage (*e.g.*, [1], [2], or to enforce security policies on the flow of sensitive user data in networked environments [19]).

Byte-level taint tracking faces significant performance challenges and a number of optimizations have been suggested in earlier work. Demand emulation and shadow page table trapping were first proposed in [5] and our system directly leverages these techniques. Neon [19] is a direct extension of the [5] approach, focusing mostly on the propagation of data labels across networked systems and thus does not make an effort to significantly improve performance. Similar in spirit to our work, the Log-Based Architecture proposed in [11] attempts to improve the performance of fine-grained infromation flow analysis through asynchrony and parallelism, but depends on a major hardware extension. Speck [10] proposes a set of techniques for parallelizing security checks (including taint tracking) on commodity hardware. Speck focuses on tracking within one user-level process and assumes OS-level support for speculative execution, while our work achieves full-system tracking using a hardware emulator.

7 Conclusion

Taint-tracking, and tracking information flow in general, have been major themes in recent systems security research. Several approaches have been proposed, but they either require redesign of OSes and applications, or are too inefficient to be deployed on commodity systems used for daily activities. PTT aims to fill that gap. We have built a byte-level whole-system taint tracking substrate using off-the-shelf open-source hypervisors and emulators, thus allowing it to be used in existing OSes and applications with minimal changes. PTT invents two novel techniques: Instrumenting tracking in the emulator at a higher abstraction level, and parallelized asynchronous taint tracking to leverage the multi-core capabilities of modern systems. These techniques, together with measures that prevent taint-explosion, make a previously untenable approach far more feasible in practice.

References

[1] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[2] M. Dalton, H. Kannan, and C. Kozyrakis. Tainting is not pointless. *SIGOPS Oper. Syst. Rev.*, 44(2):88–92, 2010.

[3] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[4] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[5] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. *SIGOPS Oper. Syst. Rev.*, 40(4):29–41, 2006.

[6] A. C. Myers. JFlow: practical mostly-static information flow control. In *POPL*, pages 228–241, New York, NY, USA, 1999. ACM.

[7] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, New York, NY, USA, 1997.

[8] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, 2008.

[9] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[10] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS XIII*, pages 308–318, New York, NY, USA, 2008. ACM.

[11] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking lifeguards. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 35–45, New York, NY, USA, 2008. ACM.

[12] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74, New York, NY, USA, 2009. ACM.

[13] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: a lightweight end-to-end system for defending against fast worms. *SIGOPS Oper. Syst. Rev.*, 41(3):115–128, 2007.

[14] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. N. Krohn, C. Frey, D. Ziegler, M. F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4), 2007.

[15] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07*, pages 116–127, New York, NY, USA, 2007. ACM.

[16] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, Big Sky, Montana, October 2009.

[17] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, Berkeley, CA, USA, 2006.

[18] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, pages 225–240, 2008.

[19] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: system support for derived data management. In *VEE '10*, pages 63–74, New York, NY, USA, 2010. ACM.