Parallelizing Security Checks on Commodity Hardware

Edmund B. Nightingale*†
Microsoft Research†
ed.nightingale@microsoft.com

Daniel Peek* Peter M. Chen*
Jason Flinn*
Dept. of EECS, University of Michigan*
{dpeek,pmchen,jflinn}@umich.edu

Abstract

Speck[1] is a system that accelerates powerful security checks on commodity hardware by executing them in parallel on multiple cores. Speck provides an infrastructure that allows *sequential* invocations of a particular security check to run in parallel without sacrificing the safety of the system. Speck creates parallelism in two ways. First, Speck decouples a security check from an application by continuing the application, using speculative execution, while the security check executes in parallel on another core. Second, Speck creates parallelism between sequential invocations of a security check by running later checks in parallel with earlier ones. Speck provides a process-level replay system to deterministically and efficiently synchronize state between a security check and the original process. We use Speck to parallelize three security checks: sensitive data analysis, on-access virus scanning, and taint propagation. Running on a 4-core and an 8-core computer, Speck improves performance 4x and 7.5x for the sensitive data analysis check, 3.3x and 2.8x for the on-access virus scanning check, and 1.6x and 2x for the taint propagation check.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability; D.4.6 [*Operating Systems*]: Security and Protection; D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Performance, Reliability, Security

Keywords Operating Systems, Security, Performance, Parallel, Speculative Execution

1. Introduction

A run-time *security check* secures a system by instrumenting an application to either detect an intrusion or prevent an attack from succeeding. Security checks occur at a variety of granularities. For instance, on-access virus scanners (Miretskiy et al. 2004) execute during each read and write to disk, call-graph modeling (Wagner and Dean 2001) executes during each system call, and security checks such as taint analysis (Newsome and Song 2005), dynamic data-flow analysis (Costa et al. 2005), and data-flow graph enforcement (Castro et al. 2006) execute before a single instruction.

[1] Speck stands for Speculative Parallel Check.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/0003. . . $5.00

To detect an attack, a security check impedes the critical path of an application by blocking it until the check completes. A security check could be executed outside the critical path, but detecting an attack after damage has been done is of little use – once an attack succeeds, it is often very difficult to undo its effects.

Unfortunately, executing powerful security checks in the critical path drastically slows down performance. For example, a security check such as taint analysis can slow down application execution by an order of magnitude or more (Newsome and Song 2005).

Therefore, security researchers who wish to guard against intrusions are faced with a conundrum. Powerful security checks may cause an application to execute too slowly, but weaker checks may not detect an attack.

This paper describes a system, called Speck, that accelerates powerful security checks on commodity hardware by executing them in parallel on multiple cores. Speck provides an infrastructure that allows *sequential* invocations of a particular security check to run in parallel without sacrificing the safety of the system.

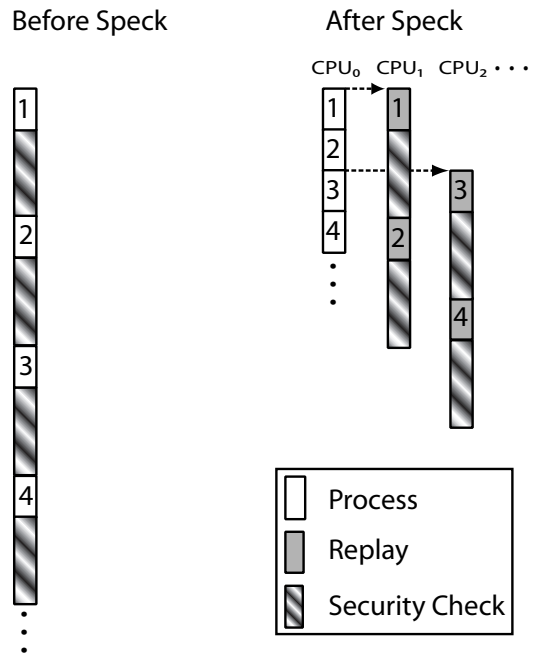
Speck creates parallelism in two ways. First, Speck decouples a security check from an application by continuing the application, using operating system support for speculative execution, while a security check executes in parallel on another core. Second, Speck takes advantage of the application executing ahead to start and run later checks in parallel to earlier ones. Thus, Speck creates parallelism not only between a process and a security check, but also between the sequential invocations of a check.

Speck uses three techniques to safely defer and parallelize security checks. First, Speck uses Speculator (Nightingale et al. 2005) to execute a process speculatively and to track and propagate its causal dependencies. Speculator allows a process to interact speculatively with the kernel, file system, and other processes. Should a security check fail, Speculator rolls back each object and process within the operating system that depends upon the compromised application.

Second, Speck buffers external output that depends on a speculative process, such as output to the network or the screen, until the security checks upon which the output depends have succeeded.

Finally, Speck provides a replay system that logs sources of non-determinism as the speculative process executes. When the security check executes in parallel, Speck replays each logged source of non-determinism to ensure that the code executing in parallel does not diverge from the speculative process.

We implemented Speck as a set of changes to the Linux 2.4 and 2.6 kernels, and we implemented parallel versions of three security checks: sensitive data analysis, on-access virus scanning, and taint propagation. Our results to date are promising. Running on a 4-core and an 8-core computer, Speck improves performance 4x and 7.5x for the sensitive data analysis check, 3.3x and 2.8x for the on-access virus scanning check, and 1.6x and 2x for the taint propagation check.



The left-hand side shows a process executing with a security check interrupting execution to check for an intrusion. The right-hand side of the figure shows how Speck parallelizes security checks. On CPU_0, the original process runs, using speculative execution, without a security check. Speck forks copies of the process, and uses replay to aggregate and execute checks in parallel on multiple cores.

Figure 1. Parallelizing a sequential security check

2. Design overview

2.1 Speck

Speck's goal is to provide security equivalent to that provided by security checks executing sequentially within the critical path of an application, while achieving better performance through parallelism. The security provided by a parallel check using Speck is equivalent to that provided by a sequential version if the output of the parallel version could have been produced by the sequential version.

The left-hand side of Figure 1 shows a program (denoted by the numbered boxes) and a security check (striped boxes) executing sequentially. The right-hand side of the figure shows how Speck parallelizes security checks on multiple cores. The process running on CPU_0 executes without a security check using operating system support for speculative execution. We refer to it as the *uninstrumented process*. The uninstrumented process invokes security checks that run in parallel on other cores. Speck replays the uninstrumented process's execution (shaded boxes) on other cores to generate the state required for each security check to run. We refer to the processes that run security checks in parallel as *instrumented clones*.

The state used by a parallel check must be identical to the state it would have used if it was executing sequentially. When it is time to begin a security check, Speck uses `fork` as a general method to copy (i.e., copy-on-write) the state of the uninstrumented process and ship it to another core. Using a software mechanism such as `fork` allows Speck to run on commodity hardware. Unfortunately, two non-trivial costs slow the start of a security check when using `fork`: first, the page table of the application must be copied and the pages must be protected. Second, during execution, both the uninstrumented and cloned process will take copy-on-write page faults.

Speck amortizes the cost of `fork` by aggregating multiple security checks into intervals of time called epochs. Increasing the epoch length reduces the frequency of `fork` and thus reduces the overhead needed to parallelize the check. However, increasing the epoch length forces the uninstrumented process to execute further ahead to begin the next epoch. For many applications, a longer epoch increases the chances that the uninstrumented process will execute a system call. Allowing the uninstrumented process to execute system calls led to a three challenges that shaped the design of Speck.

First, some system calls (e.g., `write`) allow a process to affect other processes or kernel state. It is safe to allow a speculative process to affect other processes and kernel objects as long as the affected state is rolled back when a security check fails. We use Speculator (Nightingale et al. 2005) to track and undo all changes a speculative process makes to other processes and kernel objects. Speculator tracks and propagates the causal dependencies of speculative processes as they execute. For example, if a speculative process writes to a file, that file becomes speculative and inherits the dependencies of the process. Each object and process within the kernel has an undo log associated with it. If a security check fails, Speculator uses the undo log to roll back each object and process that depends on the failed security check. Thus, a compromised program cannot permanently damage the system.

Second, other system calls allow a process to execute an output commit (output that cannot be undone, e.g., writing to the screen). When a process attempts an action that cannot be undone, the process must block until all security checks it depends on are resolved. Some output commits can be deferred, rather than blocked, which improves performance by allowing the speculative process to continue executing further ahead. For example, Speculator buffers output to the screen and the network until the output no longer depends upon speculative state. When Speculator cannot propagate a dependency or buffer output, the process is blocked until all security checks up to that point have passed.

Finally, a set of system calls exist that affect the state of the calling process (e.g., `read`). Executing one of these system calls introduces non-determinism into the execution of the uninstrumented process. The non-determinism could cause the instrumented clone to diverge from the path taken by the uninstrumented process. This could cause the check to miss an attack experienced by the uninstrumented process. Speck provides a transparent kernel-level replay system, which logs sources of non-determinism while the uninstrumented process executes ahead. Speck replays each source of non-determinism, and therefore the instrumented clone executes the same code path as its uninstrumented counterpart.

Deterministic replay enables an instrumented clone to follow the same progression of states as the uninstrumented process. Replay provides state to a subsequent security check equivalent to the state provided if the uninstrumented process calls `fork`, but at a much lower overhead. Thus, `fork` is the necessary mechanism to create parallelism, and replay is the necessary mechanism to amortize the cost of `fork` through the aggregation of security checks.

`Fork` and replay can efficiently provide the entire state of the uninstrumented process to an instrumented clone, and this allows one to run arbitrary security checks. However, some expensive security checks run infrequently or require little state to execute. In these cases, it may be faster to block the uninstrumented process and send the required state directly to a separate security check program, rather than use `fork` and replay to generate the state in an instrumented clone. For these types of checks, Speck can disable `fork` and replay and instead send only the subset of state required for the check (see Section 4.3 for an example).

2.2 Threat model

Two mechanisms ensure that Speck provides security equivalent to a sequential security check. First, due to Speck's replay system, the instrumented clones will execute the same sequential checks that would have been executed sequentially. Second, due to Speculator's causal dependency tracking and rollback, any actions a speculative process takes before the instrumented clones complete the prior checks are rolled back should a prior check fail.

Thus, Speck's security guarantee assumes that an attacker cannot compromise the replay system provided by Speck or the causal dependency tracking and rollback system provided by Speculator. Because Speck and Speculator operate within the kernel, a speculative process (which may be compromised but not yet checked) must not be allowed to gain arbitrary control of kernel state. Speculator's design philosophy helps prevent speculative processes from damaging the kernel. By default, Speculator blocks speculative processes when they make system calls; only specific system calls to specific devices are allowed to proceed. For example, Speculator allows `write` system calls only to specific devices. Thus, while the uninstrumented process may execute speculatively beyond the point of the check, it should be unable to execute actions that compromise replay or Speculator (e.g., by writing to `/dev/mem`).

We do not prevent attacks that would compromise the kernel while running a sequential version of the security check, as such attacks would compromise the system with or without Speck.

3. Speck implementation

We have implemented Speck within the Linux 2.4 and 2.6 kernels. We provide a description of the implementation of epochs, the replay system, and a brief overview of Speculator.

3.1 Creating an epoch

Speck must choose the epoch length to balance the amount of work completed by the security check and the cost of beginning a new epoch. If the epoch length is too short, the cost of `fork` dominates the benefits derived from dividing the work of the security check among multiple cores. Alternatively, if the epoch length is too long, there are fewer opportunities to create parallelism in the system, since the uninstrumented process may execute calls (e.g., `select`) that forces Speck to wait as well.

We implemented Speck to balance the cost of starting an epoch with the work accomplished during an epoch by using a time threshold. At the end of each system call, Speck checks whether the amount of time the uninstrumented program has executed is greater than a threshold epoch length; if so, Speck terminates the current epoch and forks a new instrumented clone. An instrumented clone exits when it executes the last system call in its epoch. In our experiments, we have found that an epoch length of 25–50 ms is short enough to create sufficient parallelism, but long enough to provide a significant performance improvement.

3.2 Ensuring deterministic execution

Speck uses deterministic replay to ensure that the instrumented and uninstrumented clones execute identical code paths, absent the additional security checks in the instrumented version. If the two code paths diverged, the security checks would not be valid.

When a program executes on a single core, there are only two sources of non-determinism: system calls and signal delivery. Scheduling and hardware interrupts are transparent to application execution. After a scheduling or hardware interrupt occurs, an application resumes execution at the same point that it was interrupted. Thus, operating system virtualization, which gives the application an illusion of executing alone on a virtual processor, masks this source of non-determinism from the application as long as all forms of inter-process communication are logged. Speck adds support for deterministic replay to the Linux kernel by logging system call results and signal delivery (Bressoud and Schneider 1995; Dunlap et al. 2002). Our current implementation of deterministic replay does not support shared memory IPC for multi-threaded programs running on multiple cores. Speck could be modified to support shared memory by adding support for deterministic, multiprocessor replay. Without specialized hardware support (Xu et al. 2003), Speck would need to fault on each shared memory access and record the location and effect of the fault. This technique could be prohibitively slow for workloads with many shared memory accesses.

3.3 System call replay

Speck associates a `save` and a `restore` function with each system call. When an uninstrumented program completes a system call, Speck executes the associated `save` function to log the system call results. When an instrumented clone makes a system call, Speck redirects it to a secondary *replay* system call table. Each entry in the replay system call table points to an associated `restore` function for that call, which returns the values logged by the uninstrumented version of the program. For some system calls, such as `nanosleep` and `recv`, replaying the logged value in the instrumented clone may take much less time than the original system call.

When an uninstrumented process begins a new epoch, Speck creates a *replay queue* data structure for that epoch. Each replay queue is shared between the uninstrumented and instrumented processes for a given epoch. The two processes have a producer-consumer relationship. The uninstrumented process adds system call results to the FIFO queue, and its instrumented counterpart pulls those results from the queue as it executes.

Unfortunately, this simple save and replay strategy is not sufficient for all system calls. System calls such as `read` and `gettimeofday` require special strategies because they return data by reference rather than by value, i.e., they copy data from the kernel into the user-level program's address space. To replay such calls, Speck saves the value of all user-level memory that was modified by the execution of the system call. For example, Speck saves the contents of the buffer modified by `read`. When the instrumented process calls the restore function for such system calls, Speck copies the saved values from the replay queue into the application buffer.

Currently, Speck does not support processes that execute nondeterministic functions that bypass the operating system (e.g., RAW sockets or the processor timestamp instruction `rdtsc`). Such actions could be supported either by logging them via binary instrumentation or by making the operations privileged and forcing the process to enter the kernel.

3.4 System call re-execution

Certain functions such as `mmap` and `brk` modify the layout of an application's address space. Speck must re-execute these system calls when they are called by an instrumented clone. However, Speck ensures that the modification to the instrumented version of the address space is identical to the modification made to the uninstrumented version. For example, the `mmap` system call is passed an address as a suggested starting point to map the object into its address space. If this parameter is `NULL`, then the operating system has the freedom to choose any unused point in the program's address space to map memory. Naively replaying this call might cause the operating system to map memory to two different locations in different copies of program. To prevent this problem, Speck saves the address returned by `mmap` instead of the `NULL` value of the parameter. When the instrumented clone later executes the same `mmap` system call, Speck passes the saved address into the `sys_mmap` function.

This ensures that the memory is mapped to identical locations in both copies.

3.5 Signal delivery

Linux delivers signals at three different points in process execution. First, if a process is sleeping in a system call such as `select`, Linux interrupts the sleep and delivers the signal when the process exits the kernel. Second, Linux delivers the signal when a process returns from a system call. Finally, Linux delivers a signal when a process returns from a hardware interrupt.

To ensure deterministic execution, Speck must replay the receipt of a signal at the same point in the instrumented clone as it was received by the uninstrumented process. Delivering a signal within calls such as `select` or on the return from a system call is handled by checking during system call replay whether a signal must be delivered. The delivered signals are saved in the replay queue during the execution of the uninstrumented process, and identical signals are delivered when the instrumented clone returns from the same system calls. Since the instrumented clone returns immediately from functions such as `select` rather than sleep, Speck never needs to wake up instrumented processes in the kernel to deliver signals.

Replaying signal delivery after a hardware interrupt is more difficult. To ensure correctness, a signal must be delivered at exactly the same point of execution (e.g., after some identical number of instructions have been executed). Our current implementation does not yet support this functionality, although we could use the same techniques employed by ReVirt (Dunlap et al. 2002) and Hypervisor (Bressoud and Schneider 1995) to deterministically deliver such signals. Currently, Speck simply restricts signal delivery to occur only on exit from a system call or after interrupting a system call such as `select`. In practice, this behavior has been sufficient to run our benchmarks.

3.6 OS support for speculative execution

We use Speculator (Nightingale et al. 2005) to provide support for speculative execution within the Linux kernel. Speculator can checkpoint the state of a process when it is executing within a system call and execute it speculatively.

Speculator ensures that speculative state is never visible to an external observer. If a speculative process executes a system call that would normally externalize output, Speculator buffers its output until the outcome of the speculation is decided. If a speculative process performs a system call that Speculator is unable to handle by either transferring causal dependencies or buffering output, Speculator blocks it until it becomes non-speculative.

Speck uses Speculator to isolate an untrusted application until a security check determines the application has not been compromised. Speck associates each epoch with a speculation. The success or failure of the security checks associated with an epoch determines whether the execution of code during that epoch was safe or whether an intrusion occurred. If execution was safe, Speck commits the speculation, which allows Speculator to release all output generated by that epoch. If an intrusion occurred, Speck fails the speculation, which causes Speculator to begin rollback. Each object and process dependent upon the compromised application is rolled back to a point before the intrusion occurred. We originally considered rolling back the compromised application as well, but realized that a sequential security check would likely terminate the application after detecting an intrusion. Therefore, Speculator terminates a compromised application. Since the application is never rolled back, Speculator does not checkpoint its state at the beginning of each epoch.

Currently, there is a slight difference in the output seen by an external observer when a security check fails using Speck and the output that would have been generated by a sequential security check. Since Speck operates on the granularity of an epoch, any output that occurred after the last epoch began but before the instruction that led to the failed security check would not be visible. We plan to address this by simply replaying the uninstrumented process up to the point where the security checks failed.

3.7 Pin: A dynamic binary rewriter

When Speck begins a new instrumented clone, the structure of the clone depends upon the type of security check. Speck can use any particular method of inserting checks. Two of the case studies (4.2 and 4.4) in this paper use Pin (Luk et al. 2005) to dynamically instrument the code with the necessary checks. Pin allows application programmers to create *Pintools*, which contain arbitrary code and rules about when that code should be inserted into a program binary. Pin then uses the Pintool in tandem with dynamic compilation to generate new program code.

Pintools are transparent to the execution of the application instrumented with Pin. An application instrumented with Pin observes the same addresses and registers for program code and data as it would were it running without Pin. Speck uses this property to ensure correctness: the execution of the instrumented clone does not diverge from the execution of the uninstrumented process because the program cannot observe that it is being instrumented. The authors of Pin have shown that Pin is faster than other systems such as Valgrind (Nethercote and Seward 2007) and DynamoRIO (Sullivan et al. 2003). Since Pin does not interrupt program execution unless a rule within a Pintool instructs it to do so, the overhead of Pin without a Pintool is negligible.

4. Parallelizing security checks

In this section we discuss the properties of a security check that determine both the difficulty in parallelization and the potential for performance improvement through increased parallelism. We discuss three different classes of security checks that demonstrate tradeoffs in the difficulty and overhead of parallelization and then describe the design and implementation of a check from each class.

4.1 Choosing security checks

When security checks are parallelized to run using Speck, checks that would otherwise run in sequential order execute concurrently. Some security checks may be more suitable for use with Speck (i.e., easier to parallelize) than others. This section describes the properties that determine the likelihood of success when parallelizing a check using Speck.

Speck uses the uninstrumented process to run ahead and start later security checks in parallel with earlier ones. Therefore, each security check depends upon the state received from the uninstrumented process. This dependency impacts the amount of parallelism Speck can achieve in two ways. First, the amount of work executed by the uninstrumented process between checks determines how quickly future epochs will begin. Second, the expense of the security check determines how many checks might execute in parallel at any one time. Expensive checks create more opportunity for parallelism and therefore offer the opportunity for greater relative speedup.

The benefit of parallelization also depends upon whether a later check depends upon the result of an earlier check. If a later check depends heavily on an earlier check running in parallel, it may be difficult to parallelize the check in such a way that the later check can make forward progress. If a later check is completely independent of an earlier check, the check is very easy to parallelize as the only dependency is the state provided by Speck from the uninstrumented process.

Incidentally, there is a third type of dependency relationship to consider when parallelizing a security check. Some checks may have very few dependencies within a single invocation of a check. Such a check may be embarrassingly parallel, and could be parallelized within a single invocation, rather than parallelized across multiple invocations as is done by Speck. Speck does provide one advantage, which is that the programmer does not need to reason about how to parallelize the check; parallelization is achieved automatically.

4.2 Process memory analysis

While signs of many security problems appear in the address space of a process, these signs may appear in memory only for a brief period of time. For example, a virus may decrypt itself, damage the system, then erase itself (Szor 2005); sensitive, personal data may inadvertently leak into an untrusted process (Chow et al. 2005); or a malicious insider may release classified data by encrypting it and attaching it to an e-mail. One could detect such transient problems by checking memory at each store instruction, but the overhead of such frequent checks would be prohibitive.

Speck can reduce the overhead of these checks by running them in parallel. Parallelizing this type of check with Speck is straightforward because the checks are independent of one another.

We implemented one such security check, which looks for inadvertent leaks of sensitive data. The security check carries the 16-byte hash of a sensitive piece of data. At each memory store, it calculates the hash at all 16-byte windows around the address being stored. We implemented the security check as a Pintool, which can be run sequentially or in parallel by using Speck. We used `fork` and replay to efficiently synchronize state between the security check and the uninstrumented process.

4.3 System call analysis

There have been many different proposed security checks within the research community that analyze program behavior using system calls. Examples include Wagner and Dean's (2001) call graph modeling, Provos' (2003) systrace, Ko and Redmond's (2002) noninterference check, and on-access virus scanning (Miretskiy et al. 2004). These checks can dramatically slowdown performance, which limits their usefulness in production environments. Wagner and Dean cite multiple orders-of-magnitude slowdown, and Provos notes a 4x slowdown in the worst case.

System call analysis exhibits two traits that make it promising for use with Speck. First, each check does not depend upon the result of prior checks. Therefore, later checks can be run in parallel with earlier checks without modification. Second, system call analysis requires little state to execute – often just the list of prior calls executed and/or the parameters passed by the application. This trait means that such checks do not require fork and replay; Speck can ship the state required at each system call from the uninstrumented process to an instance of the check executing in parallel.

We have implemented an on-access virus scanner using scanning libraries provided by ClamAV (2007), a popular anti-virus toolkit for Linux. Our on-access scanner intercepts file system related system calls, blocks the calling process, and passes the name of the file down to a waiting scanning daemon. Once the file has been checked, the process is allowed to continue executing.

We parallelize the check by creating a pool of waiting virus scanning daemons. Since only the name of the file is required to execute the scan, `fork` and replay are not used. Instead, our on-access scanner sends the name of the file to the next available scanner, and allows the process to continue executing speculatively.

4.4 Taint analysis

Taint analysis traces the flow of data from untrusted sources (e.g., a socket) through an application's address space and detects if a critical instruction executes with tainted (i.e., untrusted) data. For example, if the data on the top of the stack used by `RET` is tainted, a stack smash attack has occurred.

As a process executes, the taint analysis check updates a map representing which addresses and registers are tainted within the process's address space. Each check depends upon an updated version of the map. Therefore, each check depends upon all prior checks. These inter-check dependencies make taint analysis difficult to parallelize. Running multiple checks in parallel will not work, since a later check executing in parallel to an earlier check will not yet have all the information necessary to determine whether an attack has occurred.

Because taint analysis is difficult to parallelize, we use it as a challenging case study to parallelize with Speck. Our strategy is to divide work into parallel and sequential phases of execution. The parallel phase executes within instrumented clones, and the sequential phase processes the parallel pieces to determine whether an attack has occurred. Our goal is to push as much work into the parallel phase as possible, while minimizing the work done sequentially. The rest of this section gives a broad overview of instruction-level taint analysis on x86 processors.

Prior taint analysis systems (Costa et al. 2005; Newsome and Song 2005; Qin et al. 2006) use binary rewriters to instrument a target application. A memory or register location is marked as *tainted* if it causally depends on data from an external source (e.g., the disk or a network socket). For example, if an application reads data from the network, a taint tracker would mark the memory locations into which the data is copied as tainted. Data movement instructions (e.g., `MOV`, `ADD`, and `XCHG`) propagate taint from one location to another. Such instructions may also clear an address of taint either by overwriting it with untainted data or by executing an explicit clear instruction (e.g., `XOR eax, eax`). Taint trackers must set input, propagation, and assert policies to determine when data becomes tainted, which instructions propagate (or clear) taint, and when to check for an intrusion, respectively.

An *input policy* determines when data from outside the address space taints data within it. The input policy for our algorithm taints data from any source external to the process's address space, excepting shared libraries loaded by the dynamic loader. Setting an input policy can be reduced to a question of trust: if the loader and shared libraries are trusted, then loading a shared library does not taint the address space.

A *propagation policy* determines which instructions propagate taint from one location to another. For example, taint analysis functions typically instrument each data movement instruction to propagate taint. Taint analysis tools may also instrument control instructions such as `JMP`; e.g., a conditional jump based on a tainted value could be considered to taint all data values modified later. Our propagation policy instruments all data movement (`MOV, MOVS, PUSH, POP, ADD, CMOV`, etc.) instructions within the application, including F86 and SSE instructions, to propagate taint from data sources to data sinks.

There is a tradeoff in determining a propagation policy between the *coverage* provided and the *accuracy* of the algorithm. If taint is propagated along all channels of information flow, taint analysis will generate many false positives. However, if too many channels of information flow are ignored, then attacks may go undetected. Since Speck is independent of the specific taint analysis check, we chose policies from prior work (Costa et al. 2005; Newsome and Song 2005).

Finally, an *assert policy* determines when to check whether an attack has occurred. Assert policies share the same cover-

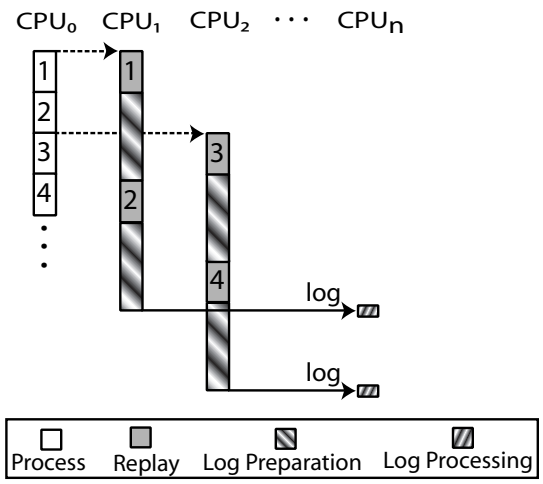


Figure 2. Parallelizing taint analysis using Speck

age/accuracy tradeoff that is inherent in propagation policies. We implemented two assert policies – checking for stack smashing attacks and function pointer attacks.

Since the sequential and parallel taint trackers required two different implementations, we divide the discussion into two separate sections. Section 5 describes our sequential taint analysis check, and Section 6 describes our parallel taint analysis check.

5. Sequential taint analysis implementation

We built a sequential taint analysis check as a baseline comparison by following the descriptions written by Costa et al. (2005) and Newsome and Song (2005).

Our sequential taint tracker uses a page table data structure to track the tainted memory addresses of the instrumented application. The top level of the page table is an array of pointers. Each entry points to a single page of memory in the instrumented application's address space. Each 4 KB page is a character array, such that each byte in the array represents a single byte of application memory. We use a byte rather a bit array because our initial results showed that bit operations substantially slowed down the taint tracker. To save space, initially each entry of the page table points to the same *zero page*, which is a 4 KB zero-filled array (indicating that the corresponding memory addresses are untainted). The taint values of registers are stored in another small array.

We use Pin to instrument each instruction that is covered by an input, propagation, or output policy. Since Pin in-lines Pintool code that is constructed without conditionals, the instrumentation that comprises the sequential taint tracker is carefully constructed to avoid conditional instructions.

Although our focus is on the benefits of parallelization, we strove to efficiently implement Newsome and Song's sequential taint tracker using Pin. Our results show that our implementation runs approximately 18 times slower than native speed; in comparison, Newsome and Song (2005) reported that their implementation ran 24-37 times slower on an identical benchmark. Based on this, we believe that our sequential implementation is reasonably efficient.

6. Parallel taint analysis implementation

The need to sequentially update a map of tainted addresses and registers as a process executes poses a challenge for Speck; running instrumented clones in parallel would result in incorrect results, since each epoch would have an incomplete map of tainted memory locations. We realized that the taint analysis check has two parts: first, the check decodes an instruction and determines whether the instruction is part of an input, propagation, or assert policy. Second, the check either updates the map of tainted memory locations, or it checks to ensure a critical instruction is not acting on tainted data.

Although updating the map of tainted locations and checking for intrusions is sequential, decoding instructions and determining whether they apply to our policies can be done in parallel. Therefore, we divided the taint analysis check into parallel and sequential phases. Figure 2 shows an example of our parallel taint tracker. In this example, each numbered box is a single instruction, and an epoch is two instructions in length (in practice, an epoch encompasses thousands of instructions). Within each instrumented clone, the instruction is replayed, and then the first phase, called *log preparation*, is executed. Log preparation transforms the instruction into a dependency operation (described in detail in Section 6.1) and places the operation into a log. The log of each epoch is divided into log segments, which are fixed sized units of memory managed by Speck. Once the epoch completes, the log is shipped to another core, where *log processing* takes place. Log processing processes each log in sequential order, updates a map of tainted addresses and registers, and checks for violations of assert policies.

One might imagine that the amount of data generated during log preparation could be quite large. In fact, our early results showed that the amount of data was so large that log processing took longer to complete than log preparation. This problem spurred the development of *dynamic taint compression*, which is often able to reduce the number of operations in a log by a factor of 6. Dynamic taint compression is designed based on two insights: first, later operations often make earlier operations unnecessary. Second, the state of the taint map is only important before an assert check and at the end of each log. We use these observations to reduce the size of each log in parallel during log preparation. A detailed explanation appears in Section 6.2. Thus, log preparation is a two step process. First, log generation (described in Section 6.1) decodes the instructions, maps them to input, propagation, or assert policies and inserts the operations into the log. Second, dynamic taint compression (described in Section 6.2) reduces the size of the log, which accelerates the sequential log processing phase.

6.1 Log generation

Log generation is implemented as a Pintool that instruments the target application. For each instruction that could potentially trigger an input, propagation, or assert policy, the Pintool writes an 8 byte log record that describes the operation type and size, the source location, and the destination location of the instruction.

Each entry in a log may be one of six types. A `TAINT` record is inserted when an input policy dictates that an address in the application's address space is tainted by its execution (e.g., TAINT records are inserted when data is read from a network socket). A `CLEAR` record is inserted if an instruction removes taint from a location. For example, the instruction `XOR eax, eax` generates a log entry that clears the `eax` register of taint. An `ASSERT` record enforces assert policies by checking to see if a location is tainted. For instance, `ASSERT eax` indicates that Speck should halt program execution if the eax register is tainted.

The remaining record types implement propagation policies. `REPLACE` records represent instructions such as `MOV` that overwrite the destination operand with the contents of the source operand. `ADD` records describe arithmetic and similar operations, where the

destination operand is tainted if either the source or destination is initially tainted. `SWAP` is used for the `XCHG` instruction, which swaps the source and destination operands. More complex operations such as `XADD` are described using multiple log records (e.g., a `SWAP` followed by an `ADD`). Operations such as `MOVS` that move data between two addresses are encoded as two separate records. Some instructions do not generate any log records since they do not affect any taint policy.

During an epoch, multiple log segments are generated to form a log. Rather than detect the end of a segment by executing an expensive conditional after inserting each log record, Speck inserts a guard page at the end of each log segment that is mapped inaccessible. After the log segment is filled, writing the next log entry causes the OS to deliver a signal to the instrumented application. The signal handler optimizes the filled log segment, as described in the next section, swaps a fresh log segment into the process address space, and changes the pointer of the next log entry to point the start of the new segment. Finally, Speck reserves a specified amount of physical memory to store log segments; each log segment is created as fixed-sized shared memory segment that is shipped from an instrumented clones to the log processing program.

6.2 Dynamic taint compression

Without optimization, it takes longer to process a log than it takes to run the sequential taint tracker described in Section 5. This is unfortunate because log processing is fundamentally sequential. The log processing program must read each log record, decode the operation, and perform the specified operation. In contrast, the sequential taint tracker has less overhead — it simply moves or verifies taint as appropriate for the instruction it is about to execute. Further, since the sequential taint tracker executes in the address space of the application it is instrumenting, it enjoys locality of reference with the addresses it is checking.

Fortunately, we have found that there is substantial room to optimize logs after they are generated but before they are processed. Since Speck speculatively executes instrumented code, it can potentially eliminate all taint operations that become moot due to later execution of the program. In contrast to static analysis tools that can only eliminate a taint operation if it becomes moot along all possible code paths, the Speck optimizer is dynamic and can eliminate an operation if it becomes moot along the code path the *application actually took*. In effect, speculative execution allows the parallel taint checker to peer into the future and eliminate work that will become unnecessary.

The goal of optimization is two-fold. First, optimization should be a parallel operation. Second, optimization should derive (and eliminate) the set of records that, after examining all records in the log segment, do not have an effect either on the final taint value of any address or register, or on the taint value of an address or register when it is checked for an assert policy violation. For example, if a register is tainted and later cleared, and no assert policy check on that register occurred between the two records, the earlier record can be safely discarded.

Dynamic taint compression is inspired by mark-and-sweep garbage collection. After each log segment is generated, it is then optimized independently of other log segments. The optimizer makes two passes over the log segment. We use a map, similar to the one described on Section 5, to track the taint values of addresses and registers. During optimization, each address or register can have one of three values; either the location is known to be tainted, known to be free of taint, or its state is unknown. If the state is unknown, it depends upon the results of prior log segments, and it might depend upon one or more records within the current segment. The second pass examines the map and identifies locations whose taint value depends upon a list of one or more records within the log segment. The optimizer then creates a new log segment, which is the union of these lists. The second pass also creates a list of ranges of tainted and free locations. After the second pass, the new, smaller log segment (and the list of tainted and free locations) is passed on to the log processing phase.

One can easily imagine more aggressive optimizations. However, we have found that the dynamic taint compression algorithm described in this section hits a sweet spot in the design space — it is very effective (achieving a 6:1 reduction in log size in our experiments), while minimally impacting performance due to its use of only two sequential passes through the log segment. Investigating other optimizations is an interesting direction for future work.

6.3 Log processing

The algorithm used in the log processing phase is very similar to that of the sequential taint tracker. It is implemented as a standalone process that reads operations from log segments and uses them to propagate and check taint, rather than as a Pintool. However, it uses identical data structures to store taint data and implements the same input, propagation, and assert policies.

To process an optimized log segment, the log processing program first reads in the set of locations known to be tainted and the set known to be taint-free. It clears and sets the taint bytes for these locations accordingly. Speck first orders log segments in the sequential order they were generated by each instrumented clone. Second, each log (the set of segments generated by an instrumented clone) is ordered sequentially as well. Segments from a later epoch cannot be processed until all segments from all earlier epochs have been processed. If any `ASSERT` record operates on tainted data, the uninstrumented program is halted and an attack reported. Otherwise, the log processing program blocks until the next sequential log segment becomes available.

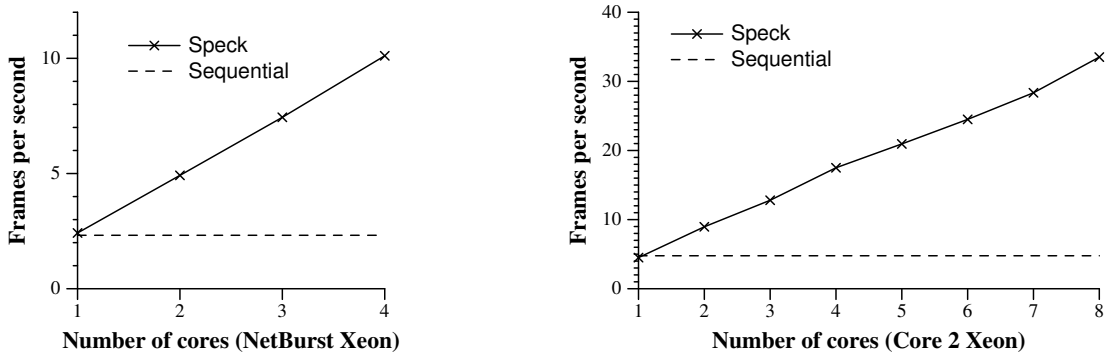
There is an inherent tradeoff in the choice of log segment size. Larger segment sizes create contention on the memory bus and pollute the caches on the system. Smaller segment sizes lead to greater overhead, since a signal handler is executed after each log is filled. Further, since the log segment is the unit of optimization, smaller log sizes reduce the efficiency of optimizations that can be performed. Our parallel taint analysis check uses 512KB logs, which are small enough to fit in the L2 cache of the computers we use in our evaluation.

7. Evaluation

7.1 Methodology

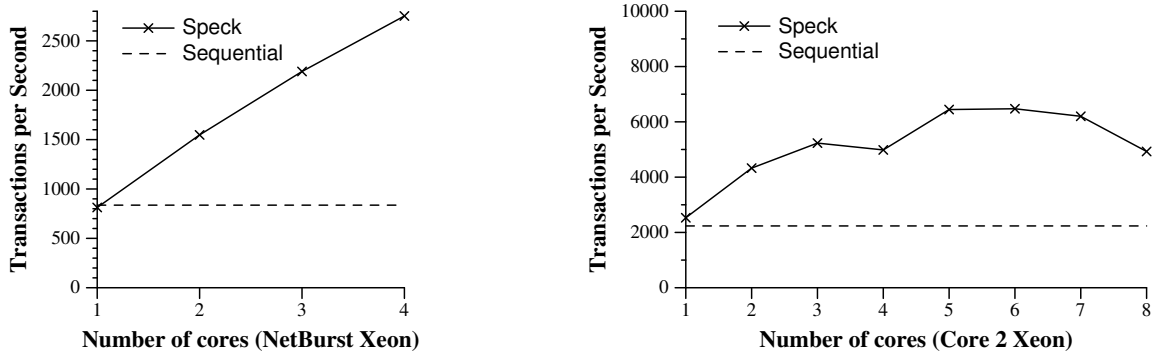
We used two computers in our evaluation. The first computer is an 8-core (dual quad-core) Intel Xeon (5300 series, Core 2 architecture) 2.66 GHz workstation with 4 GB of memory and a 1.33 GHz bus. Each quad-core chip is made up of 2 dual-core chips stacked on top of one another. Each dual-core on the chip shares a 4 MB L2 cache. The 8-core computer runs RedHat Enterprise Linux 4 (64 bit) 2.6 kernel. The second computer is a 4-core (dual dual-core) Intel Xeon (7000 series, NetBurst architecture) 2.8 GHz workstation with 3 GB of memory and a 800 MHz bus. Each dual-core chip shares a 2 MB L2 cache. The 4-core computer runs RedHat Enterprise 3 (32 bit) 2.4 kernel.

When we evaluated the parallel version of a security check, we varied the number of cores available to the operating system using the GRUB boot loader. On the 8-core computer, we could not boot any 32 bit Linux 2.4 kernel. Since Speculator currently works only with a 32 bit Linux 2.4 kernel, we could not run speculative execution for experiments on this machine. The overhead of Speculator has previously been shown to be quite small, typically adding performance overhead of a few percent or less (Nightingale et al. 2005,



This figure shows the frames per second achieved while mplayer decodes an H.264 video clip. Speck parallelizes a check that continuously scans mplayer's memory contents. The dashed line shows the performance of the sequential version of the security check. Without any security check, mplayer decodes 102 fps on the 4-core machine and 175 fps on the 8-core machine. The results represent the mean of 5 trials. Standard deviations were less than 1%. Note that the scales of the graphs differ.

Figure 3. Process memory analysis: mplayer



This figure shows the transactions per-second (TPS) executed by the Postmark benchmark as its files are scanned by a on-access virus scanner parallelized using Speck. The dashed line shows the performance of the sequential virus scanner. Without any security checks, PostMark executes 23,992 TPS on the 4-core machine and 53,078 TPS on the 8-core machine. The results represent the mean of 5 trials. Standard deviations on the 4-core machine were less than 1%. Standard deviations on the 8-core machine for 1-4 cores were less than 1%, deviations for the remaining cores were: 2.6% for 5 cores, 5.7% for 6-cores 1.8% for 7 cores and 2.8% for 8 cores. Note that the scales of the graphs differ.

Figure 4. On-access virus scanning: PostMark

2006). We confirmed this for Speck by running all benchmarks on the 4-core machine with and without Speculator.

7.2 Process memory analysis

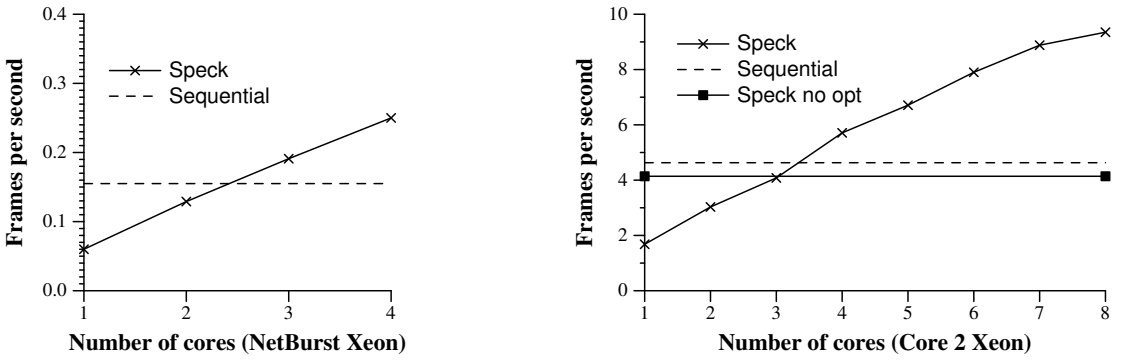
We first evaluated the performance of the process memory analysis check, described in Section 4.2, by measuring the rate at which mplayer can decode an H.264 video clip (a Harry Potter trailer) while its memory is continuously monitored for a 16-byte item of sensitive data. Without any security check, mplayer decodes the movie at a rate of 102 frames per second on the 4-core machine and 175 frames per second on the 8-core machine.

Our results are shown in Figure 3. With 8 available cores, mplayer achieves a 7.5x improvement in performance when compared to the sequential version of the check. By using Speck, the movie is decoded in real-time, which was not possible using the sequential version of the check. Speck is able to achieve near-ideal speedups for two reasons. First, the parallel version of the check is identical to the sequential version, so they perform similarly on one core. Second, the checks are independent of each other, so performance improves linearly with the number of cores.

7.3 System-call analysis

We next measured the performance of the on-access virus scanner, described in Section 4.3, by executing the PostMark (Katcher 1997) benchmark. PostMark is an I/O bound benchmark; it was designed to replicate the small file workloads seen in electronic mail, netnews, and web-based commerce. We used PostMark version 1.5, built our virus scanner using ClamAV version 0.91.2 and used a virus database containing 148,000 signatures. We configured the virus scanner to scan files on close, emulating a policy that checks for viruses as the mail or news server writes data to disk. We report the transactions per-second (TPS) achieved during the benchmark. With no security checks, PostMark executes 23,992 TPS on the 4-core machine and 53,078 TPS on the 8-core machine.

Figure 4 shows the TPS executed by PostMark with a single, sequential virus scanner, and with multiple virus scanners executing in parallel with Speck. Since the virus scanners have no dependencies between them, we expected the performance of PostMark to scale similarly to the memory analysis check. We saw this behavior on the 4-core machine; compared to the sequential virus scanner, PostMark is able to execute 3.3x more TPS (on 4 cores) by using Speck. The 8-core results were quite different. Although initially



This figure shows the frames per second achieved while mplayer decodes an H.264 video clip using the taint tracker parallelized with Speck. The dashed line shows the frames per second using the sequential taint tracker. The solid horizontal line shows the frames per second when dynamic taint compression is turned off. Without any security check, mplayer decodes 102 frames per second on the 4-core machine and 175 frames per second on the 8-core machine. The results represent the mean of 5 trials. Standard deviations were less than 1%. Note that the scales of the graphs differ.

Figure 5. Taint analysis: mplayer

scaling well (1.94x more TPS with 2 cores), performance drops with 4-cores, improves with 5 cores, and then drops again with 7 and 8 available cores. We hypothesize that the single shared memory bus may be to blame; future architectures with less contention to main memory may yield better results.

7.4 Taint analysis

Figure 5 shows the throughput of the parallel and sequential taint trackers for the mplayer video decoding benchmark. The right hand graph shows results for the 8-core machine. For one core, the total computation performed by the parallel version of the taint tracker is substantially larger than that done by the sequential version, due to the overhead of generating and optimizing log segments. However, as the number of available cores increases, the performance of the parallel version improves. With 4 cores, the parallel taint tracker outperforms the sequential version. With 8 cores, the parallel version achieves a 2x speedup compared to its sequential counterpart.

The speedup achievable through parallelization is limited by the inherently sequential nature of processing taint propagation. The 8-core graph of Figures 5 shows that optimizing the log before processing it significantly reduces this bottleneck. Without first optimizing the log, the time to run the benchmark was limited by the log processing phase, which runs on a single core. In contrast, the optimized version of taint analysis shifts enough work to the parallel phase so that Speck can effectively use additional cores.

8. Related work

Researchers in computer architecture have proposed a style of parallelization similar to ours called thread-level speculation (Knight 1986; Sohi et al. 1995; Steffan and Mowry 1998). Thread-level speculation is intended to take advantage of SMT (simultaneous multithreading) architectures (Tullsen et al. 1995) by speculatively scheduling instruction sequences of a single thread in parallel, even when those sequences may contain data dependencies. Thread-level speculation requires hardware support to quickly fork a new thread, to detect data dependencies dynamically, and to discard the results of incorrect speculative sequences. Two projects have used the proposed hardware support for thread-level speculation to parallelize checks that improved the security or reliability of software (Oplinger and Lam 2002; Zhou et al. 2004). Oplinger and Lam (2002) used this style of parallelization to speed up basic-block profiling, memory access checks, and detection of anomalous memory loads. Zhou et al. (2004) used this style of parallelization to speed up functions that monitored memory addresses (watchpoints).

In contrast to prior work on thread-level speculation, our work shows how to parallelize security checks on commodity processors. The major challenge without hardware support is amortizing the high cost of starting new threads (instrumented processes). To address this challenge, we start instrumented processes at coarse time intervals, log non-determinism to synchronize the instrumented processes with the uninstrumented process, and use OS-level speculation to enable the uninstrumented process to safely execute beyond system calls. OS-level speculation enables speculations to span a broader set of events and data than thread-level speculation, including interactions between processes and between a process and the operating system.

Patil and Fischer (1995) proposed "shadow processing", which runs an instrumented process in parallel with an uninstrumented process. Speck extends this idea to achieve more parallelism by using speculation to run the uninstrumented process ahead speculatively and allowing it to fork multiple instances. A different approach is to predict which security checks will be needed and to execute the checks speculatively (Oyama et al. 2005).

OS-level speculation has been used for many purposes besides security (Chang and Gibson 1999; Fraser and Chang 2003; Nightingale et al. 2005; Qin et al. 2005; Li et al. 2005; Nightingale et al. 2006). In the area of security, Anagnostakis et al. (2005) use speculation to improve security. They test suspicious network input in an instrumented process (a "shadow honeypot") and roll it back if it detects an attack. Speck differs from this work by parallelizing the work done by security checks, both with the uninstrumented process and with later security checks.

Researchers at Intel proposed hardware support to enable monitor checks to run in parallel with the original program (Chen et al. 2006). They proposed to modify the processor to trace an uninstrumented process and ship the trace to a monitor process running on another core. As with Speck, this enables the uninstrumented process to run in parallel with the security check. Speck differs from this work in several ways. Most importantly, Speck enables a second type of parallelism, which is to run in parallel a sequence of security checks from different points in a process's execution. Second, Speck allows the uninstrumented version to safely execute system calls that do not externalize output (rather than blocking

on every system call). Third, Speck only ships non-deterministic events to the monitoring process (rather than a trace of every instruction), and this lowers the overhead of synchronization by an order of magnitude. Finally, Speck requires no hardware support.

Finally, many researchers have explored ways to accelerate specific security checks. For example, Ho et al. (2006) accelerate taint analysis by enabling and disabling analysis as taint enters and leaves a system, and Qin et al. (2006) accelerate taint analysis through runtime optimizations. These ideas are orthogonal to our work, which seeks to accelerate arbitrary security checks by running them in parallel with the original program and with each other.

9. Conclusion

Two trends motivate our work on Speck. First, the difficulty of defending systems against attackers is spurring on the development of active defenses that use expensive run-time security checks. Second, future processor improvements will most likely come from increased parallelism due to multiple cores, rather than from substantial improvements in clock speed.

Speck stands at the confluence of these two trends. It reduces the cost of instrumenting security checks by decoupling them from the critical path of program execution, parallelizing them, and running them on multiple cores. Our results for three security checks show substantial speedups on commodity multiprocessors available today. We hope that parallelizing security checks in this manner will enable the development and use of a new class of powerful security checks.

Acknowledgments

We thank Mona Attariyan, Ya-Yunn Su, Kaushik Veeraraghavan and the anonymous reviewers for feedback on this paper. The work was supported by the National Science Foundation under award CNS-0509093. Jason Flinn was supported by NSF CAREER award CNS-0346686, Peter Chen was supported by NSF grant CNS-0614985 and Edmund Nightingale was supported by a Microsoft Research Student Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Microsoft, the University of Michigan, or the U.S. government.

References

ClamAV: The clam anti-virus toolkit, 2007. http://www.clamav.net.

K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 2005 USENIX Security Symposium*, August 2005.

Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, Copper Mountain, CO, December 1995.

Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 147–160, Seattle, WA, October 2006.

Fay Chang and Garth Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 1–14, New Orleans, LA, February 1999.

Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. *2006 Workshop on Architectural and System Support for Improving Software Dependability*, October 2006.

Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 2005 USENIX Security Symposium*, pages 331–346, August 2005.

Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 133–147, Brighton, United Kingdom, October 2005.

George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

Keir Fraser and Fay Chang. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Technical Conference*, pages 325–338, San Antonio, TX, June 2003.

Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical Taint-based Protection using Demand Emulation. In *Proceedings of EuroSys 2006*, 2006.

Jeffrey Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, 1986.

Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 177–187, May 2002.

Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Technical Conference*, 31–44, April 2005.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, , and Erez Zadok. Avfs: An on-access anti-virus file system. In *Proceedings of the 13th USENIX Security Symposium*, pages 73–88, 2004.

Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation 2007*, San Diego, CA, June 2007.

James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In Proceedings of the 12th Network and Distributed Systems Security Symposium*, February 2005.

Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 191–205, Brighton, United Kingdom, October 2005.

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 1–14, Seattle, WA, October 2006.

Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability using speculative threads. In *Proceedings of the 2002 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 184–196, October 2002.

Yoshihiro Oyama, Koichi Onoue, and Akinori Yonezawa. Speculative security checks in sandboxing systems. In *Proceedings of the 2005 International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

Harish Patil and Charles N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *Proceedings of the International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 119–132, May 1995.

Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., August 2003.

Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 235–248, Brighton, United Kingdom, October 2005.

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 *The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)*, Orlando, FL, 2006.

Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 1995 International Symposium on Computer Architecture*, June 1995.

J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 1998 Symposium on High Performance Computer Architecture*, February 1998.

G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators*, 2003.

Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 1995 International Symposium on Computer Architecture*, June 1995.

David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of 2001 IEEE Symposium on Computer Security and Privacy2001*, pages 156–169, 2001.

Min Xu, Rastislav Bodik, and Mark D. Hill. A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 2003 International Symposium on Computer Architecture*, June 2003.

Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the 2004 International Symposium on Computer Architecture*, June 2004.