An Algorithm for Detecting Contention-Based Covert Timing Channels on Shared Hardware

Jie Chen
George Washington University
Washington, DC
jiec@gwu.edu

Guru Venkataramani
George Washington University
Washington, DC
guruv@gwu.edu

ABSTRACT

As we increasingly rely on computers to process and manage our personal data, safeguarding sensitive information from malicious hackers is a fast growing concern. Among many forms of information leakage, covert timing channels operate by establishing an illegitimate communication channel between two processes and transmitting information via timing modulation, violating the underlying system's security policy. Recent studies have shown the vulnerability of popular computing environments, such as cloud, to these covert timing channels. In this work, we propose an algorithm to detect the possible presence of covert timing channels on shared hardware that use contention-based patterns for communication. Preliminary experiments demonstrate that our algorithm is able to successfully detect different types of covert timing channels at varying bandwidths, message patterns, and has zero false alarms.

1. INTRODUCTION

Information leakage is a fast growing concern affecting computer users exacerbated by the increasing amount of shared hardware resources inside the processor chip. Every year, there are hundreds of news reports on identity thefts and leaked confidential information to unauthorized parties. NIST National Vulnerability Database reports an increase of 11× in the number of information leak/disclosure-related software issues over the past five years (2008-2013), compared to the prior decade (1997-2007) [22].

Covert timing channels are information leakage channels where a trojan process intentionally modulates the timing of events on certain shared system resources to illegitimately reveal sensitive information to a spy process [26]. Note that the trojan and the spy do not communicate explicitly through send/receive or shared memory, but covertly via modulating certain events (Figure 1). In contrast to side channels where a process unintentionally leaks information to a spy process, covert timing channels have an insider trojan process (with higher privileges) that intentionally colludes with a spy process (with lower privileges) by dynamically establishing a communication protocol and exfiltrating the system secrets at runtime.

HASP '14, June 15 2014, Minneapolis, MN, USA

http://dx.doi.org/10.1145/2611765.2611766.

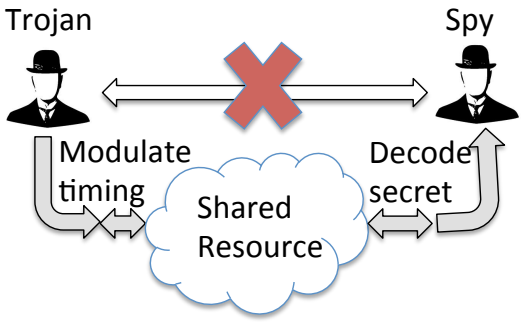


Figure 1: A Covert Timing Channel uses timing modulation on a shared resource to divulge secrets.

In order to achieve covert timing based communication on shared processor hardware, a fundamental strategy that the trojan process needs to use is modulating the timing of events through intentionally creating contention[1]*, and the spy process deciphers the secrets by observing the differences in resource access times.* On hardware units such as buses/interconnects, the trojan creates distinguishable *contention* patterns on the shared resource. Note that this basic strategy of creating contention for timing modulation has been observed in numerous covert timing channel implementations [10, 11, 23, 35, 37, 38].

In this paper, we propose an algorithm that *detects* the presence of covert timing channels by dynamically tracking contention patterns on *shared processor hardware*. We explore low-cost hardware support that gathers data on certain key indicator events during program execution, and provides software support to compute the likelihood of covert timing channel on specific shared hardware. Many prior works on covert channels have studied mitigation techniques for *specific* hardware resources [35, 10, 11, 29]. These techniques can neatly complement our detection algorithm by mitigating the damages caused by covert timing channels. Note that detection of network-based covert information transfer channels [1], software-based covert timing channels (such as data objects, file locks) [16] and side channels [2, 17, 19, 36]

[1]We use "contention" to collectively denote methods that alter either the latency of a single event or the inter-event intervals.

are beyond the scope of our work.

Our detection algorithm can be extremely beneficial to users as we transition to an era of running our applications on remote servers that host programs from many different users. Recent studies [28, 38] show how popular computing environments like cloud are vulnerable to covert timing channels. Static techniques to eliminate timing channel attacks such as program code analyses are virtually impractical to enforce on every third-party software executing on cloud, especially when most of these applications are available only as binaries. Also, adopting strict system usage policies (such as minimizing system-wide resource sharing or fuzzing system clock to reduce the possibility of covert timing channels) could adversely affect overall system performance. To overcome these issues, dynamically detecting covert channels is a much-needed first step before adopting damage control strategies like limiting resource sharing or bandwidth reduction.

In summary, the contributions of our paper are:

1. We propose to detect shared hardware-based covert timing channels by monitoring for *contentions*.

2. We design algorithms that extract recurrent (yet, sometimes noisy) contention patterns from the event train, and show our implementation in hardware and software.

3. We evaluate the efficacy of our solution using the covert timing channels on two different types of shared hardware resources, namely memory bus/QPI and integer divider. We also conduct sensitivity experiments to study low bandwidths and randomly generated message bit patterns, where we successfully detect the presence of covert timing channels.

2. UNDERSTANDING COVERT TIMING CHANNELS

Trusted Computer System Evaluation Criteria (or TCSEC, commonly referred to as The Orange Book) [6] defines covert channel as *any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy.* In particular, covert timing channels are defined as those that would allow one process to signal information to another process by modulating its own use of system resources in such a way that the change in response time observed by the second process would provide information.

Note that, between the trojan and the spy, the task of constructing a reliable covert channel is not very simple. Covert timing channels implemented on real systems take significant amounts of synchronization, confirmation and transmission time even for relatively short-length messages. As examples, (1) Okamura et al. [23] construct a memory load-based covert channel on a real system, and show that it takes 131.5 seconds just to covertly communicate 64 bits in a reliable manner achieving a bandwidth rate of 0.49 bits per second; (2) Ristenpart et al. [28] demonstrate a memory-based covert channel that achieves a bandwidth of 0.2 bits per second. This shows that the covert channels create non-negligible amounts of traffic on shared resources to accomplish their intended tasks.

TCSEC points out that a covert channel bandwidth exceeding a rate of one hundred (100) bits per second is considered "high" based on the observed data transfer rates between several kinds of computer systems. In any computer system, there are a number of relatively low-bandwidth covert channels whose existence is deeply ingrained in the system design. If bandwidth-reduction strategy to prevent covert timing channels were to be applied to all of them, it becomes an impractical task. Therefore, TCSEC points out that channels with maximum bandwidths of less than 0.1 bit per second are generally not considered to be very feasible covert timing channels. This does not mean that it is impossible to construct very low bandwidth covert timing channel, just that it becomes very expensive and difficult for the adversary (spy) to extract any meaningful information out of the system.

3. THREAT MODEL AND ASSUMPTIONS

Our threat model assumes that the trojan wants to *intentionally* communicate the secret information to the spy covertly by modulating timing on certain hardware. We assume that the spy is able to seek the services of a compromised trojan that has sufficient privileges to run inside the target system. As confinement mechanisms in software improve, hardware-based covert timing channels will become more important. So, we limit the scope of our work to shared processor hardware.

In this preliminary study, we *do not consider* cache-based covert channels that usually rely upon data storage in specific cache blocks to alter access timing and communicate data.

A covert timing channel could have noise due to two factors- (1) processes *other than* trojan and spy use the shared resource *frequently*, (2) trojan artificially inflates patterns of random conflicts excessively to evade detection. In both cases, the reliability of covert communication is severely affected resulting in loss of data for the spy as evidenced by many prior studies [24, 27, 39]. For example, Xu et al. [39] find that the covert transmission error rate is at least 20% when 64 concurrent users share the same processor with trojan/spy. Therefore, we point out that it is impossible for a covert timing channel to just randomly inflate conflict events or operate in extremely noisy environments simply to evade detection. In light of these prior findings, we model moderate amounts of interference by running a few other (at least three) active processes alongside the trojan/spy processes in our experiments.

In this work, our focus is on the detection of covert timing channels rather than showing how to actually construct or prevent them. We do not evaluate the robustness of covert communication itself that has been demonstrated adequately by prior works [28, 39, 38]. We note that prevention strategies can be appropriately deployed once the presence of covert timing channels is confirmed.

We assume that covert timing based communication happens through recurrent patterns of conflicts over non-trivial intervals of time. Our algorithm cannot detect the covert timing attacks that happen instantly where the spy gains sensitive information in one pass.

Finally, we assume that the trusted software modules (including the operating system kernel and security enforcing layers) are free of bugs and vulnerabilities (that could likely result in exposing the secrets directly).

4. DESIGN OVERVIEW

From the perspective of covert timing channels that exploit shared processor resources such as bus/interconnect

and integer divider, trojan and spy rely on patterns of high and low contention to communicate on the corresponding shared resource. Consequently, a recurrent (yet sometimes noisy) pattern of contention (conflicts) shall be observed in the corresponding event time series when the trojan covertly communicates the message bits to the spy process.

We design pattern detection algorithms to identify the recurrence patterns in the corresponding event time series. Our solution is inspired from studies in neuroscience that analyze patterns of neuronal activity to understand the physiological mechanisms associated with behavioral changes [15].

To demonstrate our algorithm's effectiveness, we use two realistic covert timing channel implementations, one of which (memory bus [38]) have been demonstrated successfully on Amazon EC2 cloud servers. We evaluate using the cycle-accurate full system simulator MARSSx86 [25] that runs Ubuntu 11.04. The simulator models a quad core processor running at 2.5 GHz, each core with two hyperthreads, and has a few (at least three) other active processes to create real system interference effects. We model a private 32 KB L1 and 256 KB L2 caches. Prior to conducting our experiments, we validated the timing behavior of our covert timing channel implementations on marss against measurements in a real system environment (dual-socket Dell T7500 server with Intel 4-core Xeon E5540 processors running at 2.5 GHz, Ubuntu 11.04). Note that the two covert timing channels described below are randomly picked to test our detection algorithm. Our algorithm is neither limited to nor derived from their specific implementations, and can be used to detect covert timing channels on all shared processor hardware using recurrent patterns of conflicts for covert communication.

4.1 Covert Timing Channels on Shared hardware

To illustrate the covert timing channels that occur on shared hardware structures and their associated indicator events, we choose memory bus and integer divider (a similar implementation was shown using multipliers by Wang et al [35]).

In case of the memory bus covert channel, when the trojan wants to transmit a '1' to the spy, it intentionally performs an atomic unaligned memory access spanning two cache lines. This action triggers a memory bus lock in the system, and puts the memory bus in contended state for most modern generations of processors including Intel Nehalem and AMD K10 family. The trojan repeats the atomic unaligned memory access pattern for a number of times to sufficiently alter the memory bus access timing for the spy to take note of the '1' value transmission. Even on x86 platforms that have recently replaced the shared memory bus with QuickPath Interconnect (QPI), the bus locking behavior is still emulated for atomic unaligned memory transactions spanning multiple cache lines [12]. Consequently, delayed interconnect access is still observable in QPI-based architectures. To communicate a '0', the trojan simply puts the memory bus in un-contended state. The spy deciphers the transmitted bits by accessing the memory bus intentionally created through cache misses. It times its memory accesses and detects the memory bus contention state by measuring the average latency. The spy accumulates a number of memory latency samples to infer the transmitted bit. Figure 2 shows the average loop execution time observed by the spy for a randomly-chosen 64-bit credit card number. A contended bus increases the memory latency making the spy to infer '1', and an un-contended bus helps the spy to infer '0'.

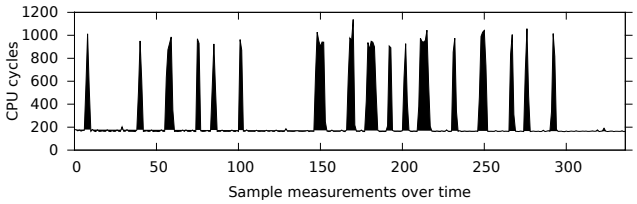


Figure 2: Average latency per memory access (in CPU cycles) in Memory Bus Covert Channel

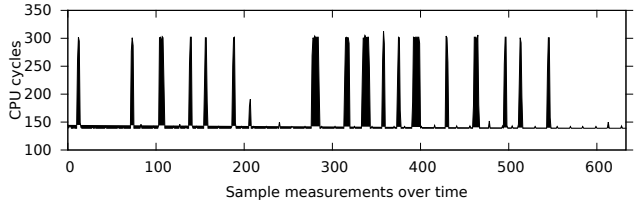


Figure 3: Average loop execution time (in CPU cycles) in Integer Divider Covert Channel

For integer divider, both the trojan and the spy processes are run on the same core as hyperthreads. The trojan communicates '1' by creating a contention on all of the division units by executing a fixed number of instructions. To transmit a '0', the trojan simply puts all of the dividers in an uncontended state by simply executing an empty loop. The spy covertly listens to the transmission by executing loop iterations with a constant number of integer division operations and timing them. A '1' is inferred on the spy side using iterations that take longer amounts of time (due to contentions on the divider unit created by the trojan), and '0' is inferred when the iterations consume shorter time. Figure 3 shows the average latency per loop iteration as observed by the spy for the same 64-bit credit card number chosen for memory bus covert channel. We observe that the loop latency is high for '1' transmission and remains low for '0' transmission.

4.2 Recurrent Burst Pattern Detection

The *first* step in detecting covert timing channels is to identify the event that is behind the hardware resource contention. In the case of memory bus covert channel, the event to be monitored is memory bus lock operation. In the case of integer division covert channel, the event to be monitored is the number of times a division instruction from one process waits on a *busy* divider occupied by an instruction from another process. Note that not all division operations fall in this category.

The *second* step is to create an *Event Train*, i.e., a unidimensional time series showing the occurrence of events (see figures 4a and 4b). We notice a large number of thick bands (or bursty patterns of events) whenever the trojan intends to covertly communicate a '1'.

As the *third* step, we analyze the event train using our recurrent burst pattern detection algorithm. This step consists of two parts: (1) check whether the event train has significant contention clusters (bursts), and (2) determine if the times series pattern exhibits recurrent patterns of bursts.

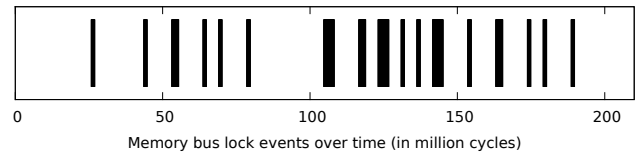
Our algorithm is as follows:

1. *Determine the interval (Δt) for a given event train to*

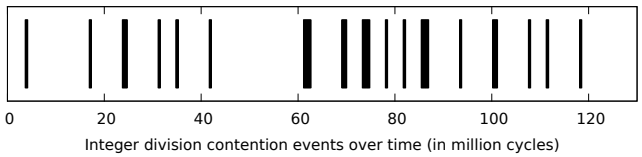
calculate event density. Δt (Equation 1) is the product of the inverse of average event rate and α, an empirical constant determined using the maximum and minimum achievable covert timing channel bandwidth rates on a given shared hardware.

$$\Delta t = \frac{1}{\frac{\#events}{acquisition\ time(sec)}} * \alpha \quad (1)$$

In simple terms, Δt is the observation window to count the number of event occurrences within that interval. The value of Δt can be picked from a wide range, and is tempered by α factor which ensures that Δt is neither too low (when the probability of a certain number of events within Δt follows Poisson distribution) nor too high (when the probability of a certain number of events within Δt follows normal distribution). For covert timing channel with memory bus, Δt is determined as 100,000 CPU cycles (or 40 μs), and in the case of covert timing channel with integer divisions, Δt is determined as 500 CPU cycles (or 200 ns).



(a) Memory Bus



(b) Integer Divider

Figure 4: Event Train plots for Memory Bus and Integer Divider showing burst patterns.

2. *Construct the event density histogram using Δt.* For each interval of Δt, the number of events are computed, and an event density histogram is constructed to subsequently estimate the probability distribution of event density. An illustration is shown in Figure 5. The x-axis in the histogram plot shows the range of Δt bins that have a certain number of events. Low density bins are to the left, and as we move right, we see the bins with higher numbers of events. The y-axis shows the number of Δt's within each bin.

3. *Detect burst patterns.* From left to right in the histogram, threshold density is the first bin which is smaller than the preceding bin, and equal or smaller than the next bin. If there is no such bin, then the bin at which the slope of fitted curve becomes gentle is considered as the threshold density. If the event train has burst patterns, there will be two distinct distributions- (1) one where the mean is below 1.0 showing the non-bursty periods, and (2) one where the mean is above 1.0 showing the bursty periods present in the right tail of the event density histogram. Figure 6 shows the event density histogram distributions for covert timing channels involving bursty contention patterns on memory bus and integer dividerr. For both timing channels, we see significant non-burst patterns in the histogram bin# 0. In case of memory bus channel, we see significant bursty pattern at histogram bin#20. For Integer Division, we see a very prominent second distribution (burst pattern) between bins#84 and #100 with peak around bin#90.

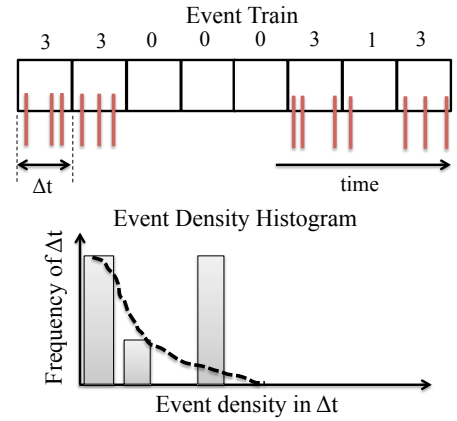
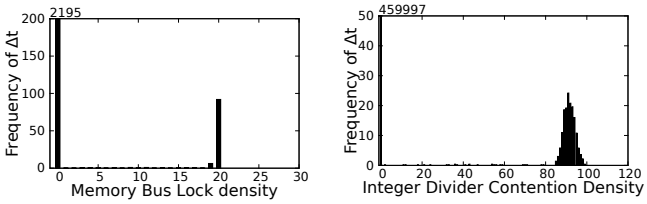


Figure 5: Illustration of Event Train and its corresponding Event Density Histogram. The distribution is compared against the Poisson Distribution shown by the dotted line to detect presence of burst patterns.



(a) Memory Bus (b) Integer Divider

Figure 6: Event Density Histograms for Covert Timing Channels using Memory Bus and Integer Divider.

4. *Identify significant burst patterns (contention clusters) and filter noise.* To estimate the significance of burst distribution and filter random (noise) distributions, we compute the likelihood ratio[2] of the second distribution. Empirically, based on observing realistic covert timing channels [38, 29], we find that the likelihood ratio of the burst pattern distribution tends to be at least 0.9 (even on very low bandwidth covert channels such as 0.1 bps). On the flipside, we observe this likelihood ratio to be less than 0.25 among regular programs that have no known covert timing channels despite having some bursty access patterns. We set a conservative threshold for likelihood ratio at 0.5, i.e., all event density histograms with likelihood ratios above 0.5 are considered for further analysis.

5. *Determine the recurrence of burst patterns.* Once the presence of significant burst patterns are identified in the event series, the next step is to check for recurrent patterns of bursts. We limit the *window of observation* to 512 OS time quanta (or 51.2 secs, assuming a time quantum of 0.1 secs), to avoid diluting the significance of event density histograms involved in covert timing channels. We develop a pattern clustering algorithm that performs two

[2]Likelihood ratio is defined as the number of samples in the identified distribution divided by the total number of samples in the population [21]. We omit bin#0 from this computation since it does not contribute to any contention.

basic steps- (1) discretize the event density histograms into strings, and (2) use k-means clustering to aggregate similar strings. By analyzing the clusters that represent event density histograms with significant bursts, we can find the extent to which burst patterns recur, and hence detect the possible presence of covert timing channel. We note that our algorithm can detect covert timing channels *regardless* of burst intervals (i.e., even on low-bandwidth bursts or random noise due to interference from the system environment), since it uses clustering to extract recurring burst patterns.

5. IMPLEMENTATION

For implementation, we explore a hardware-software cooperative approach, where the hardware gathers the key indicator events and the software analyzes the patterns to detect the possibility of covert timing channels. In this section, we show the hardware modifications and software support that shall be needed by our algorithm.

5.1 Hardware Support

In current microprocessor architectures, we note that most hardware units are shared by multiple threads, especially with the widespread adoption of Simultaneous Multithreading (SMT) support. Therefore, all of the microarchitectual units are potential candidates to be a medium for covert timing channel.

For preliminary implementation, we design a dedicated *auditor* hardware that has the ability to *randomly audit* any microarchitectual unit. To minimize implementation complexity, at any given time, we assume that the auditor unit selects a limited number (say, two) of hardware units to monitor. Note that the auditor does not have any fundamental limitation in monitoring more than two hardware units at a time. The philosophy of auditing a small number of hardware units or instructions have been commonly adopted by most runtime (performance) monitors simply to minimize design cost [8, 7, 30].

The Instruction Set is augmented with a special instruction that lets the user to program the auditor and choose the certain hardware units to audit. This special instruction shall be a privileged instruction that only a subset of system users (usually the system administrator) can utilize for system monitoring. The hardware units have a programmable bit, which when set, places the hardware unit under audit for covert timing channels. The hardware units are wired to fire a signal to the auditor on the occurrence of certain key indicator events seen in covert timing channels.

For most of the core components such as execution clusters and logic, the indicator events are conflicts detected by a hardware context when another context is already using them. On certain uncore components such as memory bus, conflicts are created using special events such as bus locking that trigger the signal to the auditor.

To accumulate the event signals arriving from the hardware units, the auditor contains (1) a 32-bit count-down register initialized to the value of Δt, (2) two 16-bit register to accumulate the number of event occurences within Δt, and (3) two histogram buffers with 128 entries (each entry is 16-bits long) to record the event density histograms. Whenever the event signal arrives from the unit under audit, the accumulator register is incremented by one. At the end of each Δt, the two 16-bit accumulator values are updated against the corresponding entry in the histogram buffers, and the count-down register is reset to Δt. At the end of OS time quantum, the histogram buffers are recorded by the software module.

5.1.1 Area, Latency and Power Estimates

We use Cacti 5.3 [14] to estimate the area, latency and power needed for our auditor hardware. Table 1 shows the results of our experiments. For the two histogram buffer, we model 128-entries that are each 16-bits long. For registers, we model two 128-byte vector registers, two 16-bit accumulators, and one 4-byte countdown register. Overall, we note that our area overheads are insignificant compared to the total chip area (e.g., 263 mm^2 for Intel i7 processors [13]). The auditor hardware has latencies that are less than processor clock cycle time (0.33 ns for 3 GHz). Given that the auditor hardware is accessed only when conflicts happen, it is unlikely that the auditor hardware would extend the clock cycle period. Similarly, the dynamic power drawn by auditor hardware are in the order of a few milliwatts compared to 130 W peak in Intel i7 processors [13].

Table 1: Area, Power and Latency Estimates

	Histogram Buffers	Registers
Area(mm^2)	0.0028	0.0011
Power(mW)	2.8	0.8
Latency(ns)	0.17	0.17

5.2 Software Support

In order to place a microarchitectural unit under audit, the user requests the auditor through a special software API exported by the OS. The OS is responsible for privilege checks before letting the user to monitor the unit.

A separate daemon process (part of our software support) accumulates the data points by recording the histogram buffer contents at each OS time quantum (for contention-based channels). Lightweight code is carefully added to avoid perturbing the system state, and to record performance counters as accurately as possible [5]. To further reduce perturbation effects, the OS scheduler could be made to schedule the processes on currently un-audited cores.

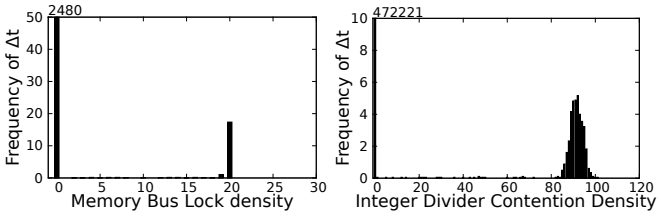
Since our analysis algorithms are run as background processes, they incur minimal effect on system performance. Our pattern clustering algorithm is invoked every 51.2 secs (Section 4.2) and takes 0.25 secs (worst case) per computation. We note that further optimizations such as feature dimension reduction reduces the clustering computation time to 0.02 secs (worst case).

6. EVALUATION AND SENSITIVITY STUDY

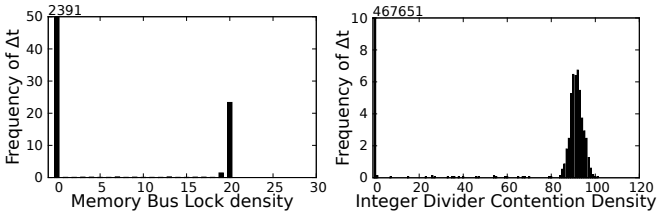
First, we test our detection algorithm by varying the bandwidth rates and message bit patterns. Second, we test our algorithm on a variety of standard memory- and CPU-intensive benchmarks. We evaluate using *combinations of* I/O-intensive Filebench server benchmarks [9], memory-intensive Stream [20] and SPEC2006 CPU benchmarks [31] with reference inputs. For each run, we use *taskset* command to pin our benchmarks onto two hyperthreads that share the same physical core. These second set of experiments are to evaluate our algorithms' behavior on applications with no known covert channel, and to test whether the normal burst patterns usually observed in such benchmarks trigger any false alarms.

6.1 Varying Bandwidth Rates

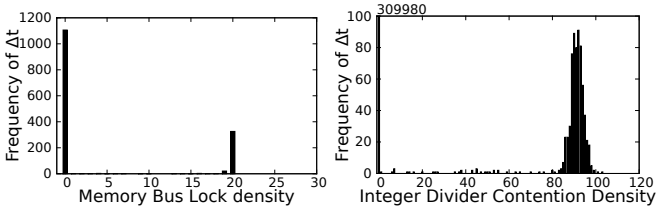
We conduct experiments by altering the bandwidth rates of three different covert timing channels from 0.1 bps to 1000 bps. The results (observed over a window of OS time quantum, 0.1 secs) are shown in Figure 7. While the magnitudes of Δt frequencies decrease for lower bandwidth contention-based channels, the likelihood ratios for second (burst) distribution are still significant (higher than 0.9)[3].



(a) Bandwidth=0.1bps



(b) Bandwidth=10bps



(c) Bandwidth=1000bps

Figure 7: Bandwidth test using Memory Bus and Integer Divider

6.2 Encoded Message patterns

To simulate encoded message patterns that the trojan may use to transmit messages, we generate 256 random 64-bit combinations, and using them as inputs to the covert timing channels. Our experimental results are shown in Figure 8. Mean values of histogram bins are shown by dark bars that are annotated by the range (maximum, minimum) of bin values observed across the 128 runs. Despite variations in peak magnitudes of Δt frequencies (especially in integer divider), we notice that our algorithm still shows significant second distributions with likelihood ratios above 0.9.

6.3 Testing for False Alarms

We test our recurrent burst algorithms on 128 pair-wise combinations of several standard SPEC2006, Stream and

[3]The histogram bins for second distribution (covert transmission) are determined by the number of successive conflicts needed to reliably transmit a bit and the timing characteristics of the specific hardware resource. For example, Δt for memory bus channel is 100,000 cycles and minimum inter-access interval between successive conflicts is 5,000 CPU cycles. Therefore, the second distribution is clustered around bin#20.

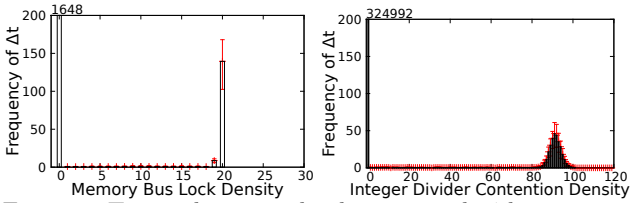


Figure 8: Test with 256 randomly generated 64-bit messages on Memory Bus and Integer Divider. Black (thick) bars are the means, and the red (annotations) arrows above them show the range (min, max).

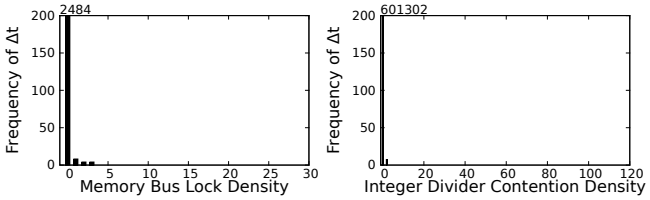
Filebench benchmarks run simultaneously on the same physical core as hyperthreads. The SPEC2006 applications are run in random pairs. Stream and Filebench benchmarks are run as cloned copies pinned onto a single core as hyperthreads. Specifically, we pick two different types of servers from Filebench- (1) webserver, that emulates web-server I/O activity producing a sequence of open-read-close on multiple files in a directory tree plus a log file append (100 threads are used by default), (2) mailserver, that mail server that stores each e-mail in a separate file consisting of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single directory (16 threads are used by default). Despite having some regular bursts and conflict cache misses, all of benchmark pairs are known to not have any covert timing channels. Due to space constraints, we are unable to show all of our experimental results. Figure 9 presents a representative subset of our experiments. We observe that most of the benchmark pairs have either zero or random burst patterns for both memory bus lock (first column) and integer division contention (second column) events. The only exception is mailserver pairs, where we observe a second distribution with bursty patterns between histogram bins#5 and #8. Upon further examination, we find that the likelihood ratios for these distributions was less than 0.25 (which is significantly less than the ratios seen in all of our covert timing channel experiments). Therefore, we did not observe any false alarms in our recurrent burst pattern detection algorithms.

7. RELATED WORK

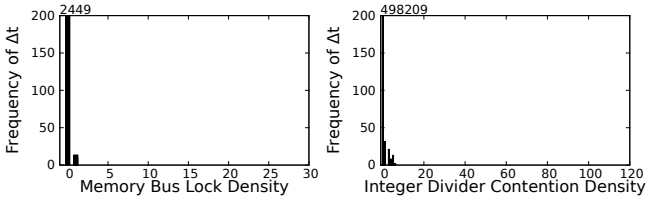
The notion of covert channel was first introduced by Lampson et al [18]. Hu et al [11] proposed fuzzing system clock by randomizing interrupt timer period between 1ms and 19 ms. Unfortunately, this approach could significantly affect system's normal bandwidth and performance in the absence of any covert timing channel activity. Among studies that consider processor-based covert timing channels, Wang et al. [35] identify two new covert channels using exceptions on speculative load (*ld.s*) instructions and SMT/multiplier unit. Wu et al. [38] present a high-bandwidth and reliable covert channel attack that is based on QPI lock mechanism where they demonstrate their results on Amazon's EC2 virtualized environment. Ristenpart et al. [28] present a method of creating cross-VM covert channel by exploiting the L2 cache, which adopts the Prime + Trigger + Probe [32] to measure the timing difference in accessing two pre-selected cache sets and decipher the covert bit. Our algorithm is tested using examples derived from prior covert timing channel implementations on shared hardware.

To detect and prevent covert timing channels, Kemmerer et al. [16] proposed a shared matrix methodology to *statically* check whether potential covert communications could happen using shared resources. Wang et al [34] propose a covert channel model for an abstract system specification. Unfortunately, such static code-level or abstract model analyses are impractical on every single third-party application executing on a variety of machine configurations in today's computing environments, especially when most of these applications are available in binary-only format.

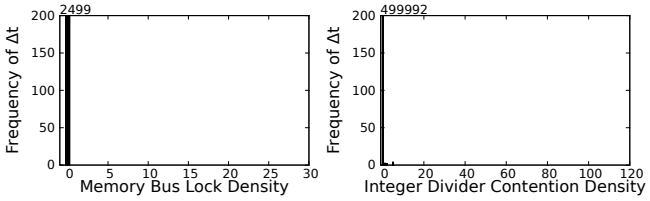
Other works have studied side channels and solutions to minimize information leakage. Side channels are information leakage mechanisms where a certain malware secretly profiles a legitimate application (via differential power, intentional fault injection etc.) to obtain sensitive information. Wang et al. [36, 33] propose two special hardware cache designs, Partition-Locking (PL), Random Permutation (RP) and New cache to defend against cache-based side channel attacks. Kong et al. [17] show how secure software can use the PL cache. Martin et al. [19] propose changes to the infrastructure (timekeeping and performance counters) typically used by side channels such that it becomes difficult for the attackers to derive meaningful clues from architectural events. Demme et al. [3] introduce a metric called Side Channel Vulnerability Factor (SVF) to quantify the level of difficulty for exploiting a particular system to gain side channel information. Coppens et al [2] use compiler transformations to obfuscate key dependent control behavior in branches and mitigate side channel attacks. Many of the above preventative techniques neatly complement our detection algorithm, in that they serve to provide enhanced security to the system.



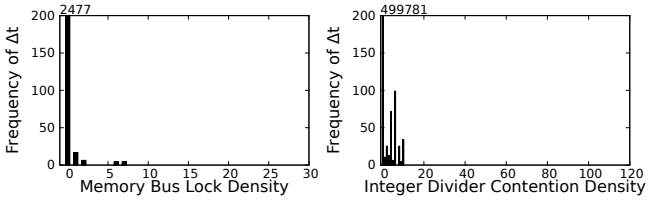
(a) gobmk_sjeng



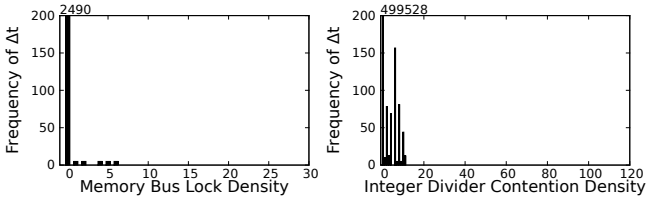
(b) bzip2_h264ref



(c) stream_stream



(d) mailserver_mailserver



(e) webserver_webserver

Figure 9: Event Density Histograms in pair combinations of SPEC2006, Stream & Filebench

Demme et al [4] have explored simple performance counters for malware analysis. This strategy is not applicable for a number of covert channels because they use specific timing events to modulate hardware resources that may not be measurable through the current performance counter infrastructure. For instance, the integer divider channel should track cycles where one thread waits for another (unsupported by current hardware). Using simple performance counters as alternatives will only lead to high number of false positives. Second, our algorithm understands the time modulation characteristics in covert timing channel implementations. Using machine learning classifiers without considering covert timing channel-specific behavior could result in high number of false alarms.

8. CONCLUSION AND FUTURE WORK

In this paper, we present an algorithm to detect the possible presence of covert timing channels on shared processor hardware. Our algorithm works by detecting recurrent burst on certain key indicator events associated with the covert timing channels. We test the efficacy of our solution using example covert timing channels on two different types of processor hardware- memory bus/QPI and integer divider. We conduct sensitivity studies by altering the bandwidth rates and message bit combinations. Through experiments on I/O, memory, CPU-intensive benchmarks such as Filebench [9], SPEC2006 [31] and Stream [20] that are known to have no covert channels, we show that our framework does not have any false alarms.

As future work, we will explore how to incorporate detection mechanisms for cache-based covert timing channels, and efficiently integrate the respective mitigation techniques after detecting covert timing channels on hardware.

9. ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under CAREER Award CCF-1149557.

10. REFERENCES

[1] S. Cabuk, C. E. Brodley, and C. Shields. Ip covert channel detection. *ACM Transactions on Information and System Security (TISSEC)*, 12(4):22, 2009.

[2] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, 2009.

[3] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *Proceedings of ISCA*, 2012.

[4] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of ISCA*, 2013.

[5] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *International Symposium on Computer Architecture (ISCA)*, pages 353–364. IEEE, 2011.

[6] Department of Defense Standard. *Trusted Computer System Evaluation Criteria DoD 5200.28-STD*. US Department of Defense, 1983.

[7] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, volume 5, 2001.

[8] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices, Inc*, 2007.

[9] File system and Storage Lab. Filebench. *http://sourceforge.net/apps/mediawiki/filebench*, 2011.

[10] J. Gray III. On introducing noise into the bus-contention channel. In *IEEE Computer Society Symposium on Security and Privacy*, 1993.

[11] W.-M. Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3):233–254, 1992.

[12] Intel Corporation. Intel 7500 chipset. *Datasheet*, 2010.

[13] Intel Corporation. Intel core i7-920 processor. *http://ark.intel.com/Product.aspx?id=37147*, 2010.

[14] N. P. Jouppi et al. Cacti 5.1. *http://quid.hpl.hp.com:9081/cacti/*, 2008.

[15] Y. Kaneoke and J. Vitek. Burst and oscillation as disparate neuronal properties. *Journal of neuroscience methods*, 68(2):211–223, 1996.

[16] R. A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems (TOCS)*, 1(3):256–277, 1983.

[17] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of HPCA*, 2009.

[18] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10), Oct. 1973.

[19] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of ISCA*, 2012.

[20] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

[21] National Institute of Standards and Technology. Maximum Likelihood, 2013.

[22] National Institute of Standards and Technology. National Vulnerability Database, 2013.

[23] K. Okamura and Y. Oyama. Load-based covert channels between xen virtual machines. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 2010.

[24] H. Okhravi, S. Bak, and S. King. Design, implementation and evaluation of covert channel attacks. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

[25] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011*, 2011.

[26] C. Percival. Cache missing for fun and profit, 2005.

[27] N. E. Proctor and P. G. Neumann. Architectural implications of covert channels. In *Proceedings of the Fifteenth National Computer Security Conference*, volume 13, pages 28–43, 1992.

[28] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[29] B. Saltaformaggio, D. Xu, and X. Zhang. Busmonitor: A hypervisor-based solution for memory bus covert channels. *EUROSEC*, 2013.

[30] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[31] Standard Performance Evaluation Corporation. Spec 2006 benchmark suite. *http://www.spec.org*, 2006.

[32] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(2), Jan. 2010.

[33] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM International Symposium on Microarchitecture*, 2008.

[34] Z. Wang and R. B. Lee. New constructive approach to covert channel modeling and channel capacity estimation. In *Proceedings of the 8th International Conference on Information Security*, ISC'05, 2005.

[35] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference*, pages 473–482. IEEE, 2006.

[36] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[37] J. C. Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3):219–232, 1992.

[38] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, 2012.

[39] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, 2011.