Detecting Hardware Covert Timing Channels

Many popular computing environments are vulnerable to covert timing channels. With improvements in software confinement mechanisms, shared processor hardware structures will be natural targets for such an attack. The authors present a microarchitecture-level framework that detects the possible presence of covert timing channels on shared hardware. Experimental results demonstrate their ability to detect different types of covert timing channels on various hardware structures and communication patterns.

Guru Venkataramani
Jie Chen
Miloš Doroslovački
George Washington University

Covert timing channels are information leakage channels in which a Trojan process intentionally modulates the timing of events on a shared system resource to illegitimately reveal data secrets to a spy process. Note that the Trojan and the spy do not communicate explicitly through send/receive or shared memory, but covertly via modulating certain hardware events. In contrast to side channels, wherein 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) to exfiltrate the system secrets.

To achieve covert-timing-based communication on shared processor hardware, a fundamental strategy used by the Trojan process is to modulate the timing of events by intentionally creating conflicts. We use "conflict" to collectively denote methods that alter either a single event's latency or the interevent intervals. The spy process deciphers the secrets by observing the differences in resource access times. On computational logic and buses and interconnects, the Trojan creates conflicts by introducing distinguishable contention patterns. On memory structures, the Trojan creates conflicts through repetitive patterns of intentional memory block replacements such that the spy can decipher the message bits based on the memory hit and miss latencies. This basic strategy of creating conflicts for timing modulation has been observed in numerous covert timing channel implementations.[1–5]

In this article, we present a framework that detects hardware covert timing channels by dynamically tracking conflict patterns on shared processor hardware. We design low-cost hardware support that dynamically gathers data on certain key indicator events on shared hardware devices, and use software support to compute the likelihood of covert timing channels operating on them. (For

0272-1732/16/$33.00 © 2016 IEEE Published by the IEEE Computer Society

Related Work in Covert Channels

Butler Lampson first introduced the notion of a covert channel.[1] Wei-Ming Hu proposed fuzzing the system clock that could significantly affect the normal system performance.[2] Richard Kemmerer proposed a shared matrix methodology to statically check whether a resource has potential for covert activity.[3] Unfortunately, such static code-level or abstract model analyses are impractical on every single third-party application in today's computing environments.

Zhenghong Wang and Ruby Lee proposed secure hardware cache designs to defend against cache side channels.[4] John Demme and colleagues introduced a metric called the *side-channel vulnerability factor* to quantify the difficulty level to exploit a system for side channels.[5] Andrew Ferraiuolo and colleagues designed a secure memory scheduling algorithm to protect the shared memory controller.[6] Hassan Wassel and colleagues proposed a new data transmission for on-chip networks to prevent information leakage.[7] Many of these preventative techniques complement our design and provide enhanced system security.

Casen Hunger and colleagues studied the same underlying phenomenon—that cache covert channels are created through contention.[8] Although they observed the destructive read property and proposed anomaly detection as the solution, CC-Hunter instead detects contention channels by analyzing the event trains generated specifically from contention events, such as conflict misses for a cache channel.[9] Dmitry Evtyushkin and colleagues used the Trojan to warm up the branch predictor such that the spy can decipher the bit based on the branch misprediction latency.[10] In essence, an oscillatory pattern of conflicts happens between the Trojan and the spy on the branch history buffer, in which the Trojan intentionally flips the predictor bits (similar to the cache block replacement done by the Trojan to create a conflict miss). With additional hardware support to track such conflicts on branch history buffer entries, our algorithm (described in the "Understanding Cache Conflict Miss Patterns" section of the main article) can be adapted to detect timing channels on branch predictors.

References

1. B.W. Lampson, "A Note on the Confinement Problem," *Comm. ACM*, vol. 16, no. 10, 1973, pp. 613–615.
2. W.-M. Hu, "Reducing Timing Channels with Fuzzy Time," *J. Computer Security*, vol. 1, no. 3, 1992, pp. 233–254.
3. R.A. Kemmerer, "Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels," *ACM Trans. Computer Systems*, vol. 1, no. 3, 1983, pp. 256–277.
4. Z. Wang and R. Lee, "New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks," *Proc. 34th Ann. Int'l Symp. Computer Architecture*, 2007, pp. 494–505.
5. J. Demme et al., "Side-Channel Vulnerability Factor: A Metric for Measuring Information Leakage," *Proc. 12th Ann. Int'l Symp. Computer Architecture*, 2012, pp. 106–117.
6. A. Ferraiuolo et al., "Lattice Priority Scheduling: Low-Overhead Timing Channel Protection for a Shared Memory Controller," to be published in *Proc. Int'l Symp. High-Performance Computer Architecture*, 2016.
7. H.M. Wassel et al., "SurfNoC: A Low Latency and Provably Non-Interfering Approach to Secure Networks-on-Chip," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013, pp. 583–594.
8. C. Hunger et al., "Understanding Contention-Based Channels and Using Them for Defense," *Proc. IEEE 21st Int'l Symp. High Performance Computer Architecture*, 2015, pp. 639–650.
9. J. Chen and G. Venkataramani, "CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware," *Proc. 47th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2014, pp. 216–228.
10. D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and Mitigating Covert Channels through Branch Predictors," *ACM Trans. Architecture and Code Optimization*, vol. 13, no. 1, 2016, doi:10.1145/2870636.

information on other approaches, see the "Related Work in Covert Channels" sidebar.)

Our framework can protect users from sensitive information leakage as we transition to an era of running applications on remote servers that host programs from many different users. Prior studies show how popular computing environments such as cloud computing are vulnerable to covert timing channels.[4,5] Static techniques to eliminate timing channels, such as code analyses, are virtually impractical to enforce on every third-party software application and on code binaries. Furthermore, adopting strict system usage policies could adversely affect the overall system performance. To overcome these issues, our dynamic detection is a desirable first step before adopting damage-control strategies.

This article offers new technical contributions over our prior work, CC-Hunter.[6] First, we design algorithms that denoise the system activity log to identify the key indicator events responsible for covert timing transmission. We also incorporate a low-cost cache conflict miss detector to replace the generation-bit-based detector in CC-Hunter. Finally, we present additional experimental results that show different communication protocols in cache covert timing channels and demonstrate the efficacy of our solution.

Defining Covert Timing Channels

Trusted Computer System Evaluation Criteria (TCSEC), also called the *Orange Book,* defines a covert timing channel as one 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.[7]

Note that, between the Trojan and the spy, the task of constructing a reliable covert timing channel is not simple. Covert timing channels implemented on real systems require significant amounts of synchronization, confirmation, and transmission time, even for relatively short messages. For example, Keisuke Okamura and Yoshihiro Oyama constructed a memory-load-based covert channel on a real system and showed 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 (bps).[3] Thomas Ristenpart and colleagues demonstrate a memory-based covert channel that achieves a bandwidth of 0.2 bps.[4] This shows that the covert channels create nonnegligible amounts of traffic on shared resources to accomplish their intended tasks.

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

Threat Model and Assumptions

Our threat model assumes that the Trojan wants to intentionally and covertly communicate the secret information to the spy through recurrent conflict patterns to modulate the timing on certain hardware over nontrivial amounts of time. We do not consider covert channels based on networks or software layers and side channels.

A hardware-based covert timing channel could have noise due to two factors: processes other than the Trojan and spy using the shared resource frequently, and the Trojan artificially inflating the patterns of random conflicts to evade detection. In both cases, the reliability of covert communication is severely affected, resulting in loss of data for the spy.[4,8] Therefore, it is impossible for a covert timing channel to randomly inflate conflict events or operate in noisy environments simply to evade detection. In light of these prior findings, we model moderate amounts of interference by running a few (at least three) active processes alongside the Trojan and spy processes in our experiments.

In this article, we use two different, realistic covert timing channel implementations on interconnect (memory bus) and memory (level-2, or L2, cache) that have been demonstrated successfully on Amazon EC2 cloud servers.[4,5]

Memory Bus Covert Timing 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 a contended state for most modern generations of processors, including the Intel Nehalem and the AMD K10 family. The Trojan repeats the atomic unaligned memory access pattern several times to sufficiently alter the memory bus access timing for the spy to note the 1 value transmission. To communicate a 0, the Trojan simply puts the memory bus in an uncontended state. The spy deciphers the transmitted bits by accessing the memory bus intentionally through main memory accesses. It times its memory accesses and detects the memory bus contention state by measuring the average latency. The spy accumulates several memory latency samples to infer the transmitted bit.

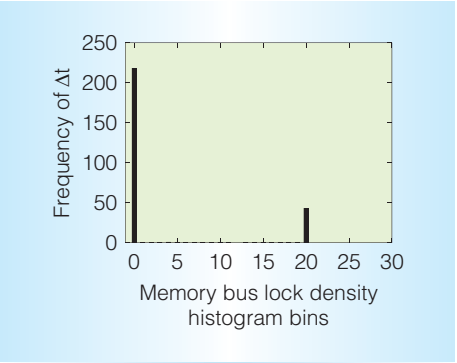

Figure 1. Event density histogram for memory bus channel demonstrating the memory bus lock density values measured within Δt intervals and their corresponding frequency of occurrence during runtime.

L2 Cache Covert Timing Channel

To transmit a 1, the Trojan visits the even-numbered cache sets (G_1) and replaces all of the constituent cache blocks; for a 0, it visits the odd-numbered cache sets (G_0) and replaces all of the constituent cache blocks. The spy infers the transmitted bits as follows: It replaces all of the cache blocks in G_1 and G_0, and times the accesses to the G_1 and G_0 sets separately. If the accesses to G_1 sets take longer than the G_0 sets, the spy infers 1. Otherwise, it infers 0.

Detection Algorithms

We use the examples described earlier to illustrate our design approach. Note that our algorithms are neither limited to nor derived from these specific examples. Our algorithms seek to detect and analyze the fundamental property exploited by hardware covert timing channels—namely, conflict patterns on indicator events.

Detecting Recurrent Burst Patterns

To detect covert timing channels, we first need to identify the event that is behind the hardware resource contention. In our example, the event to be monitored is the memory bus lock operation. Next, we need to create an *event train*—that is, a unidimensional time series showing the occurrence of events. Third, we should analyze the event train using our recurrent burst pattern detection algorithm.

Our algorithm comprises five steps.

Determine the interval (Δt) for a given event train to calculate event density. Δt is the product of the inverse of the 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 device.

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. Low-density bins are to the left, and as we move right, we see the bins with higher numbers of events within the observation interval of Δt.

Detect burst patterns. From left to right in the histogram, threshold density is the first bin that is smaller than the preceding bin, and equal to or smaller than the next bin. If there is no such bin, then the bin at which the slope of the fitted curve becomes gentle is considered as the threshold density. If the event train has burst patterns, there will be two distinct distributions (see Figure 1): one in which the mean is below 1.0, showing the nonbursty periods; and one in which the mean is above 1.0, showing the bursty periods present in the right tail of the event density histogram.

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 of the second distribution. (The likelihood ratio is defined as the number of samples in the identified distribution divided by the total number of samples in the population. We omitted bin #0 from this computation because it does not contribute to any contention.) Empirically, we find that the likelihood ratio of the burst pattern distribution tends to be at least 0.9 (even on low-bandwidth covert channels such as 0.1 bps), and less than 0.5 among regular programs that have no known covert timing channels despite having some bursty access patterns.

Determine the recurrence of burst patterns. We develop a pattern-clustering algorithm that performs two basic steps: discretize the event density histograms into strings, and 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 a covert timing channel.

Understanding Cache Conflict Miss Patterns

Unlike combinational structures in which timing modulation is performed by varying the interevent intervals (observed as bursts and nonbursts), cache-based covert timing channels rely on the latency of events to perform timing modulation. The Trojan and the spy create a sufficient number of conflict events (cache misses) alternatively among each other that let the spy decipher the transmitted bits based on the average memory access times (hits and misses). This leads to repetitive patterns of cache-conflict misses between the Trojan and spy contexts.

As we noted earlier, establishing reliable covert timing channels involves significant overhead for both the Trojan and the spy processes to first synchronize and then transmit bits covertly from the Trojan to the spy with a high reliance on noninterference from other concurrently running processes in the system.[3,4,8] Consequently, covert transmission phases typically account for nontrivial time intervals during which the spy and the Trojan create numerous cache conflict miss patterns alternatively among each other. Therefore, analyzing the cache conflict miss event train can provide insight into the possible presence of cache covert timing channel activity.

Oscillation is defined as a property of periodicity in an event train. This is different from bursts that are specific periods of high-frequency event occurrences in the event train. Oscillation of an event train is detected by measuring its autocorrelation, or the correlation coefficient of the signal with a time-lagged version of itself—that is, the correlation coefficient between two values of the same variable, X_i and X_{i+p} separated by lag p.

An autocorrelogram is a chart showing the autocorrelation coefficient values for a sequence of lag values. An oscillation pattern is inferred when the autocorrelation coefficient shows significant periodicity with peaks sufficiently high for certain lag values.

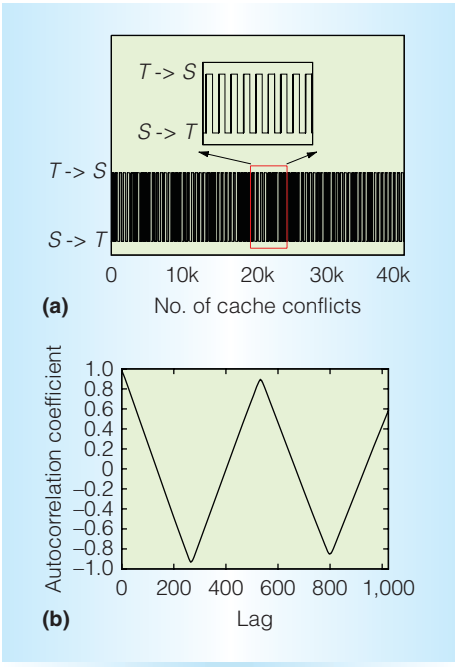

Figure 2. Level-2 (L2) cache conflict miss pattern between the Trojan (*T*) and the spy (*S*). (a) The event train. $T \rightarrow S$ shows the Trojan's conflict misses with the spy; $S \rightarrow T$ shows the spy's conflict misses with the Trojan. (b) The event train's autocorrelogram.

Figure 2 shows our conflict miss event train analysis. In particular, Figure 2a shows the event train (cache conflict misses), annotated by whether the conflicts happen due to the Trojan replacing the spy's cache sets or vice versa (the inset figure shows a legible version of the cluttered event train pattern). $T \rightarrow S$ denotes the Trojan (T) replacing the spy's (S's) blocks because the spy had previously displaced those same blocks owned by the Trojan at that time.

Note that every ordered pair of Trojan/spy contexts has unique identifiers. For example, $S \rightarrow T$ is assigned 0, and $T \rightarrow S$ is assigned 1. The autocorrelation function is computed on this conflict miss event train. Figure 2b shows the event train's autocorrelogram. A total of 512 cache sets were used in G_1 and G_0 for transmission of 1 or 0 bit values. We observe

that, at a lag value of 533 (which is close to the actual number of conflicting sets in the shared cache, 512), the autocorrelation value is highest at about 0.893. The slight offset from the actual number of conflicting sets was observed due to random conflict misses in the surrounding code and the interference from conflict misses due to other active contexts sharing the cache.

Implementation

Here, we show how to obtain events of interest from the system activity log, and we then describe the hardware modifications and software support to implement our framework.

Denoising System Events

During system runtime, we register the event parameters relating to timing channel activity that occurs within microarchitectural units. This produces event data sequences that should be further denoised to extract features for decision making about the possible presence of a covert communication.

Registration. There are two types of usage for the shared resource: processes that use the resource normally, and those that exploit the resource to perform covert communication. To track the shared resource usage and ultimately detect the possibility for covert communication, our detection framework counts the number of accesses within a time period Δt (that is, access intensity) and continues counting for nonoverlapping intervals of size Δt. This produces a sequence of access intensities that is used in analysis to detect the presence of covert communication processes. Additionally, access durations and origins of access requests are recorded. We use a small and a big aggregation period Δt simultaneously. In this way, we have resource access characterization at two different time scales. The small-time-scale analysis shows details of access dynamics, whereas the big-time-scale analysis extracts trends (that is, long-term dependencies).

Denoising. Many different processes use the shared resources simultaneously, and it is important to filter the sequences produced by processes that are not involved in any known covert timing-based communication. There are two possible situations: The covert timing activity produces sufficiently distinct signals and thus regular activities can be considered as noise, or the covert timing activity blends itself with regular activities and thus it becomes hard to distinguish its signals. To isolate access bursts from regular activity that produces low-intensity signals, we use the soft-limiter thresholding technique for noise reduction.[9] Soft limiter zeros the values below the specified threshold, whereas the threshold value trims the values above the threshold. Our detection framework determines the optimal threshold that provides the best recovery based on the knowledge of statistical properties of noise and the shapes that should be cleaned out. For processes that blend into normal system activity for certain periods, we note that they still exhibit periodicity and thus can be searched through Fourier techniques (such as bispectrum).

Hardware and Software Support

The instruction set architecture is augmented with a privileged instruction that lets the user program certain hardware units to audit. The hardware units have an *audit bit* that, when set, instructs them to fire a signal to the monitor upon certain key event occurrences.

Although it is desirable to monitor all hardware units, doing so would require smart time-multiplexing of our detector that could quickly become a performance bottleneck if all of the hardware units were monitored simultaneously. Alternatively, the use of multiple instances of hardware detectors to accommodate the monitoring of all hardware units can be cost-prohibitive. Therefore, our design provides the capability to monitor up to two different hardware units at any time. A privileged administrator-user is responsible for choosing the hardware units to monitor on the basis of his or her knowledge of the currently running applications. We believe that this design tradeoff can prevent unnecessary overheads on most regular user applications.

To accumulate the event signals arriving from the hardware units, the monitor contains two 32-bit countdown registers initialized to Δt values, two 16-bit registers to accumulate the number of event occurrences within Δt, and two histogram buffers with

128 entries (16 bit-entry) to record the event density histograms. At the end of each Δt, the corresponding 16-bit accumulator value is updated against its entry in the histogram buffer, and the countdown register is reset to Δt. At the end of the OS time quantum, the software module records the histogram buffers.

A conflict miss happens in a set associative cache when several blocks map into the same cache set and replace each other even when there is enough capacity left in the cache. To efficiently track the conflict misses in hardware, we incorporate a separate hardware buffer that records the cache block tags that get replaced from the cache. If an incoming cache block tag matches an entry in the hardware buffer (storing the recently replaced cache block tags), this denotes that the incoming cache block was prematurely replaced for cache conflict reasons. Therefore, a cache conflict miss is detected. The number of entries in this conflict-miss-tracking hardware buffer is equal to the total number of cache sets, where the buffer entries are directly mapped with one entry per cache set. Jamison Collins and Dean Tullsen showed that the accuracy of classifying conflict misses from capacity misses remains unchanged beyond maintaining 10 or more partial cache tag bits.[10] In our design, we use a 1-Kbyte buffer that holds 16-entry partial tags for the 512 L2 cache sets.

We design two alternating 128-byte vector registers that record the three-bit context IDs of the replacer and the victim. When one vector register is full, the other vector register begins to record the data. Meanwhile, the software module logs the vector contents in the background and then clears it for future use.

Our experimental results using Cacti 5.3 show that the hardware area overheads are insignificant (less than 0.01 mm^2) with access latencies less than the processor clock cycle time (0.4 ns for 2.5 GHz).[11] Also, the dynamic power drawn by the monitor is approximately 3 to 4 mW.

As part of the software support, a separate daemon process accumulates the data points by recording the histogram buffer contents at each OS time quantum (for contention-based channels) or the 128-byte vector register (for oscillation-based channels). Lightweight code is carefully added to avoid perturbing the system state and to record performance counters as accurately as possible. The daemon processes are scheduled on (currently) unaudited cores to minimize perturbation effects.

Evaluation and Sensitivity Study

Our experimentation platform includes a full system environment by booting a Micro-Architectural and System Simulator for x86-based Systems (MARSSx86[12]) with Ubuntu 11.04. The simulator models a quad-core processor running at 2.5 GHz and has at least three other active processes to create real system-interference effects. We model a private 32-Kbyte, level-1 (L1), four-way cache and a shared 256-Kbyte, L2, eight-way cache with 64-byte blocks. Prior to conducting our experiments, we validated the timing behavior of our covert channel implementations running on a MARSSx86 against the timing measurements in a real system environment (dual-socket Dell T7500 server with Intel 4-core Xeon E5540 processors at 2.5 GHz, Ubuntu 11.04).

Varying Bandwidth Rates

We conducted experiments by altering the bandwidth rates of two covert timing channels from 0.1 bps to 1,000 bps. Figure 3 shows the results (observed over a window of OS time quantum, 0.1 seconds). Although the magnitudes of Δt frequencies decrease for lower-bandwidth memory bus channels, the likelihood ratios for second (burst) distribution are still significant (higher than 0.9). On low-bandwidth cache covert channels such as 0.1 bps, despite observing periodicity in autocorrelation values, we note that their magnitudes do not show significant strength. As we reduced the sizes of the observation window to less than the OS time quanta, the autocorrelograms started to show significant repetitive peaks for the 0.1-bps channel. Our experiments suggest that autocorrelation analysis at finer granularity observation windows can detect lower-bandwidth channels more effectively.

Testing for False Alarms

We tested our recurrent burst and oscillation pattern algorithms on 128 pairwise combinations of several standard SPEC2006 (www.

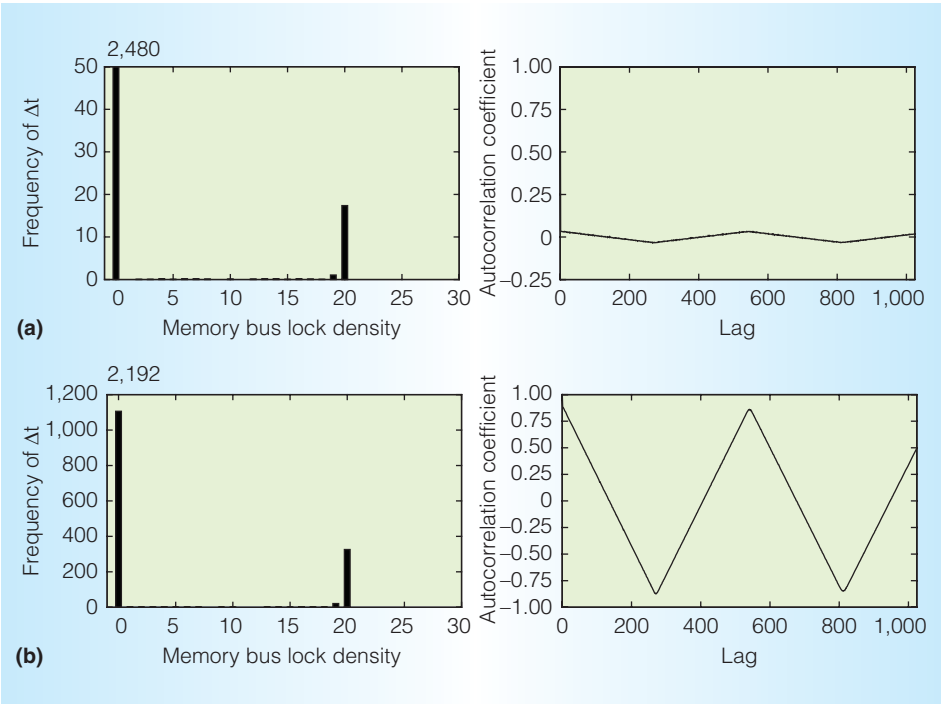

Figure 3. Bandwidth test using memory bus and L2 cache covert channels. (a) Bandwidth rate of 0.1 bits per second (bps). (b) Bandwidth rate of 1,000 bps.

spec.org), Stream,[13] and Filebench (http://sourceforge.net/apps/mediawiki/filebench) benchmarks run simultaneously on the same physical core as hyperthreads. We picked two types of servers from Filebench: webserver and mail server. We chose the individual benchmarks on the basis of their CPU-intensive (SPEC2006) and memory- and I/O-intensive (Stream and Filebench) behavior, and we paired them in such a way as to maximize the chances of them creating conflicts on a particular microarchitectural unit. Our goal was to study whether these benchmark pairs create similar levels of recurrent bursts or oscillatory patterns of conflicts that were observed in realistic covert channel implementations (which, if true, could potentially lead to a false alarm). Figure 4 presents a representative subset of our experiments. Most benchmark pairs have either zero or random burst patterns for both memory bus lock events. The only exception is the mail server pairs, wherein we observe a second distribution with bursty patterns between histogram bins #5 and #8, but the likelihood ratio is less than 0.5. In all of the autocorrelograms, we did not observe the noticeable periodicity typically expected of timing channels. Therefore, we did not observe any false alarms.

Varying Cache Channel Attacks

Earlier, we showed a cache timing channel that uses two groups of cache sets, odd (G_0) and even (G_1). The Trojan creates cache conflict misses on one group based on the bit to be transmitted, and the spy deciphers the bit by measuring the access latency differences to G_0 and G_1. In real-world situations, the attacker could choose any combination of cache sets to form groups and construct covert timing channels.

To test our detection scheme's robustness, we demonstrate three variations of cache channel attacks:

- MOD-3, in which G_0 includes cache sets whose set number is divisible by 3 and G_1 includes cache sets whose set number is not a multiple of 3. This can be generalized to set number $i \ mod \ N = 0$, where $N \geq 2$. In fact, prior work[4] uses $N = 2$.
- PRIME, in which G_0 includes prime-numbered cache sets and G_1

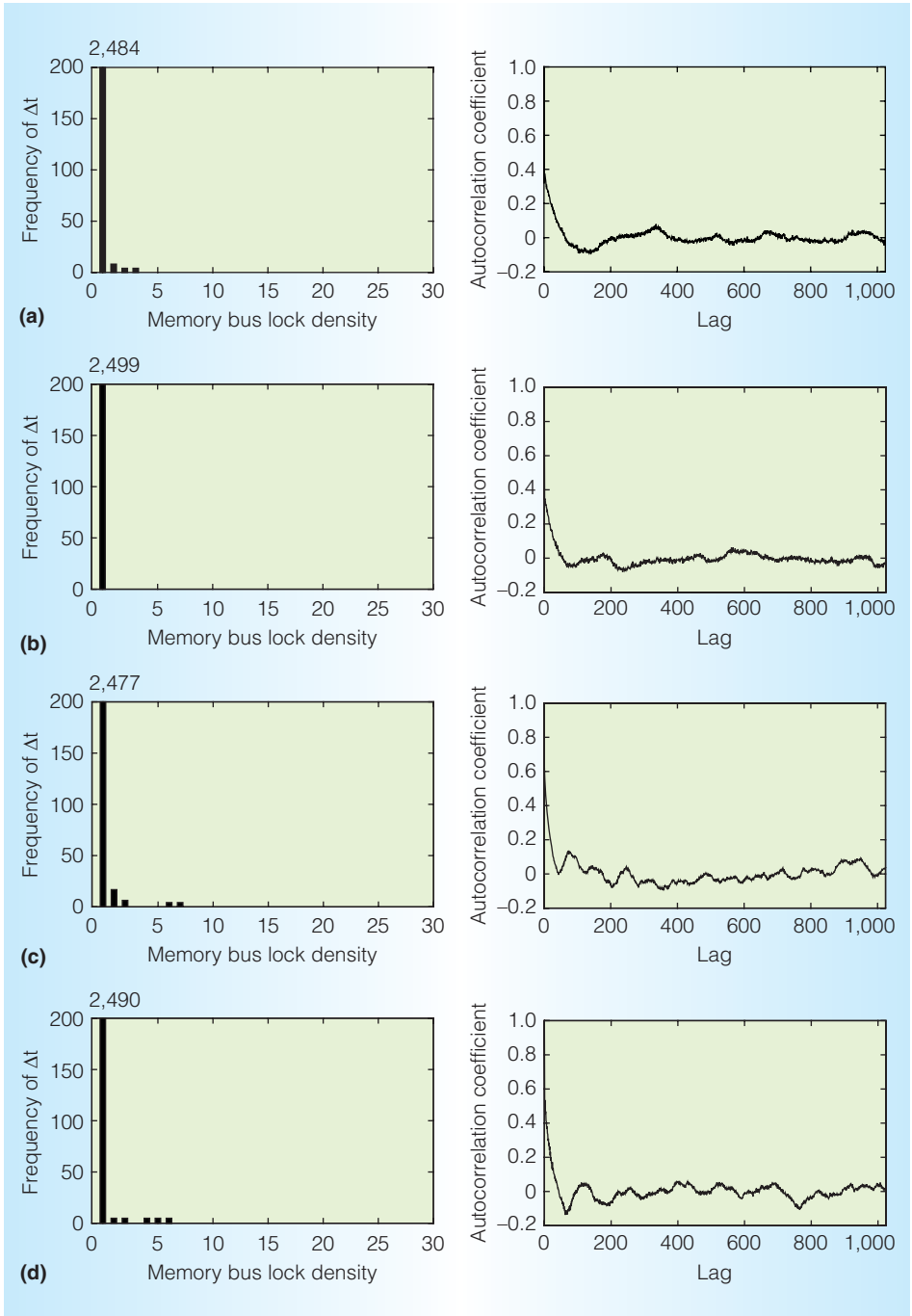

Figure 4. Event density histograms and autocorrelograms in pair combinations of SPEC2006, Stream, and Filebench. (a) gobmk_sjeng. (b) stream_stream. (c) mailserver_mailserver. (d) webserver_webserver.

includes nonprime-numbered cache sets.

- RANDOM64, in which each of G_0 and G_1 include 64 cache sets that are randomly selected by the Trojan and spy processes prior to the start of cache covert channel communication.

Figure 5 shows our results; we see that the autocorrelograms exhibit significant periodicity

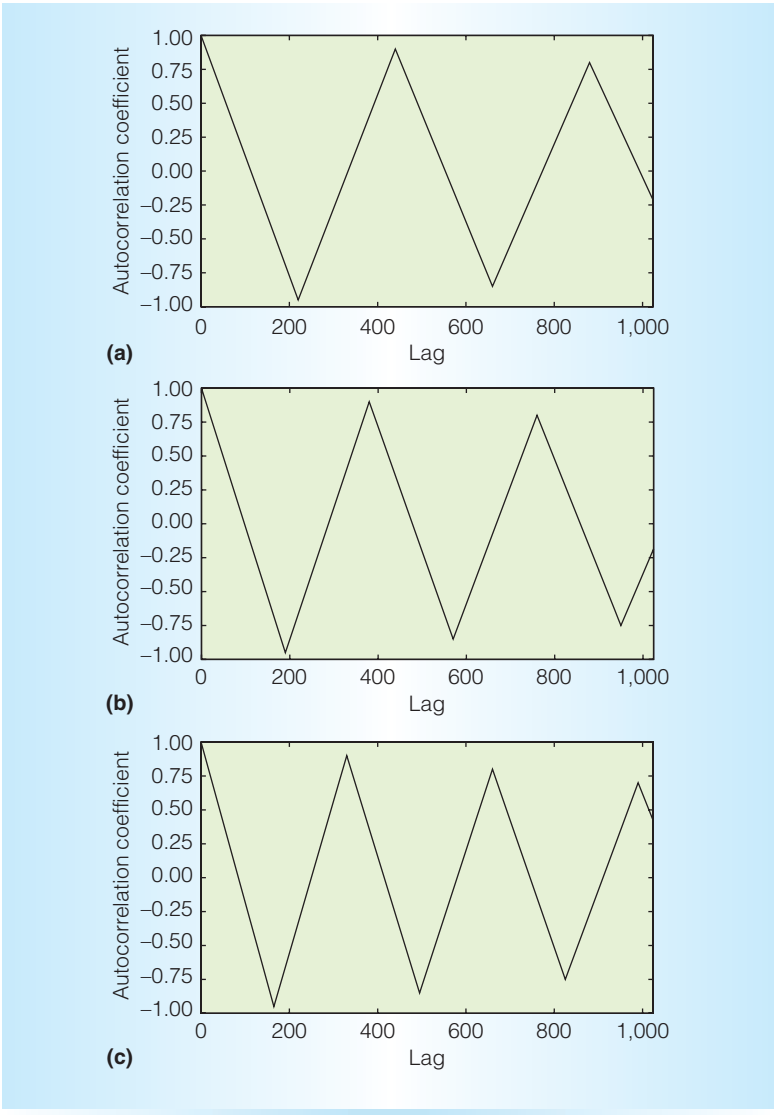

Figure 5. Autocorrelograms for cache covert channels with varying combinations of cache sets. (a) MOD-3. (b) PRIME. (c) RANDOM64.

with magnitude above 0.90. This shows that our algorithm works well irrespective of how attackers pick their cache sets, because we track oscillatory patterns of conflict misses, a fundamental characteristic of cache covert timing channels.

In this work, we have explored the first steps toward detecting the hardware covert timing channels that will be increasingly exploited by malicious users in the future to exfiltrate sensitive secrets, especially with rapid advancements in software confinement mechanisms. As defense strategies are deployed, adversaries may find unscrupulous ways to subvert them by exploiting more sophisticated timing channels involving multiple hardware devices and hardware-software interfaces. Furthermore, emerging platforms like heterogeneous architectures present the malicious users with abundant resources for exploitation. Such channels can be more challenging, because several parts of the system (possibly, hardware and software) may need to be monitored, resulting in unacceptable application runtime overheads. Subsequently, more lightweight and effective detection strategies are needed. As future work, we plan to explore such complex systems and sophisticated implementations of covert timing channels. MICRO

Acknowledgments

This material is based on work supported by the US National Science Foundation under CAREER Award CCF-1149557 and CNS-1618786, and Semiconductor Research Corp. (SRC) contract 2016-TS-2684. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of the NSF or SRC.

References

1. J. Gray III, "On Introducing Noise into the Bus-Contention Channel," *IEEE Computer Society Symp. Security and Privacy*, 1993, pp. 90–98.
2. W.-M. Hu, "Reducing Timing Channels with Fuzzy Time," *J. Computer Security*, vol. 1, no. 3, 1992, pp. 233–254.
3. K. Okamura and Y. Oyama, "Load-Based Covert Channels between Xen Virtual Machines," *Proc. ACM Symp. Applied Computing*, 2010, pp. 173–180.
4. T. Ristenpart et al., "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," *Proc. 16th ACM Conf. Computer and Comm. Security*, 2009, pp. 199–212.
5. Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-Space: High-Speed Covert Channel Attacks in the Cloud," *Proc. USENIX Conf. Security Symp.*, 2012, article 9.

6. J. Chen and G. Venkataramani, "CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware," *Proc. 47th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2014, pp. 216–228.
7. *Trusted Computer System Evaluation Criteria*, US Dept. of Defense, 1983.
8. H. Okhravi, S. Bak, and S. King, "Design, Implementation and Evaluation of Covert Channel Attacks," *Proc. IEEE Int'l Conf. Technologies for Homeland Security*, 2010, pp. 481–487.
9. S. Mallat, *A Wavelet Tour of Signal Processing: The Sparse Way*, Academic Press, 2009.
10. J.D. Collins and D.M. Tullsen, "Hardware Identification of Cache Conflict Misses," *Proc. 32nd Ann. ACM/IEEE Int'l Symp. Microarchitecture*, 1999, pp. 126–135.
11. S. Thoziyoor et al., *CACTI 5.1*, tech. report HPL-2008-20, HP Labs, Apr. 2008.
12. A. Patel et al., "MARSSx86: A Full System Simulator for x86 CPUs," *Proc. 48th Design Automation Conf.*, 2011; doi:10.1145/2024724.2024954.
13. J.D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Technical Committee Computer Architecture Newsletter*, Nov. 1995.

Guru Venkataramani is an associate professor in the Department of Electrical and Computer Engineering at George Washington University. His research interests include computer architecture, hardware support for debugging, security, and many-core computing. Venkataramani received a PhD in computer science from Georgia Tech. He is a senior member of IEEE and ACM. Contact him at guruv@gwu.edu.

Jie Chen is a senior performance engineer at MathWorks. His research interests include computer architecture, application performance, power efficiency, memory system reliability, and hardware security. Chen received a PhD in computer engineering from George Washington University, where he performed the work for this article. Contact him at jiec@gwmail.gwu.edu.

Miloš Doroslovački is an associate professor in the Department of Electrical and Computer Engineering at George Washington University. His research interests include adaptive signal processing, communications, and distributed estimation. Doroslovački received a PhD in electrical engineering from the University of Cincinnati. He is a member of IEEE and EURASIP. Contact him at doroslov@gwu.edu.