ReplayConfusion: Detecting Cache-based Covert Channel Attacks Using Record and Replay*

Mengjia Yan, Yasser Shalabi, and Josep Torrellas
University of Illinois, Urbana-Champaign
http://iacoma.cs.uiuc.edu

***Abstract*—Cache-based covert channel attacks use highly-tuned shared-cache conflict misses to pass information from a trojan to a spy process. Detecting such attacks is very challenging. State of the art detection mechanisms do not consider the general characteristics of such attacks and, instead, focus on specific communication protocols. As a result, they fail to detect attacks using different protocols and, hence, have limited coverage.**

In this paper, we make the following observation about these attacks: not only are the malicious accesses highly tuned to the mapping of addresses to the caches; they also follow a distinctive cadence as bits are being received. Changing the mapping of addresses to the caches substantially disrupts the conflict miss patterns, but retains the cadence. This is in contrast to benign programs.

Based on this observation, we propose a novel, high-coverage approach to detect cache-based covert channel attacks. It is called *ReplayConfusion*, and is based on Record and deterministic Replay (RnR). After a program's execution is recorded, it is deterministically replayed using a different mapping of addresses to the caches. We then analyze the difference between the cache miss rate timelines of the two runs. If the difference function is both sizable and exhibits a periodic pattern, it indicates that there is an attack. This paper also introduces a new taxonomy of cache-based covert channel attacks, and shows that *ReplayConfusion* uncovers examples from all the categories. Finally, *ReplayConfusion* only needs simple hardware.

I. Introduction

Cloud computing has become ubiquitous, as both companies and users employ it to minimize infrastructure cost. In conjunction with the virtualization paradigm, this has resulted in an increased sharing of hardware resources among different users. Multiple users often execute code and store private data on the same physical machine. This lends to the possibility of a malicious user crafting a subtle attack to steal private information from another user that shares the physical machine. Such attacks utilize covert communication channels that are built on top of contended shared resources. These channels can leak secrets without leaving a trace, and they cannot be regulated by a user's security policy. This makes them very effective.

A *Covert Channel* is defined as a communication channel where a sender encodes information by modulating some condition (e.g., availability of some service, timing of events, or cache conflicts) that is detectable by a receiver. Such a channel can be utilized by a trojan process to communicate sensitive information to a spy process that shares some physical resource. Covert channel attacks have been demonstrated using various shared resources, such as file system objects [1], network stacks/channels [2], [3], input devices [4], and caches and/or micro-architectural structures [5], [6], [7], [8], [9].

A common type of covert channel attack uses shared caches. These cache-based attacks consists of the trojan creating a pattern of conflicts in the shared cache that can be interpreted by the spy process [5], [9], [6], [10], [11], [8], [7]. Typically, the spy process follows the so-called *Prime+Probe* technique. During prime, the spy fills selected cache sets with its own data. Then it turns idle, and the trojan either evicts the spy's data or leaves it in the cache — depending on the message it wishes to send. Later, the spy conducts a probe by accessing the same addresses, and learns the trojan's decision by estimating the number of misses to these addresses.

To defend against cache-based covert channel attacks, several approaches have been proposed [12], [13], [14], [15], [16], [17], [18]. These techniques protect the system by either providing isolation between processes in the caches, or introducing noise in system to prevent accurate estimation of the misses. However, they are insufficient, as they either fail to thwart all possible covert channel attacks, or they introduce unacceptable performance overheads. For example, current cache partitioning techniques can only support a limited number of partitions with reasonable overhead.

An alternative approach to deal with cache-based covert channel attacks is to detect covert channel communication, and block future information leakage. Since covert channel attacks target long-term information transfer, this approach is valuable even if the detection happens after some information has already leaked. Unfortunately, there is no general, low-overhead detection solution for cache-based covert channel attacks. State of the art techniques do not consider the general characteristics of such attacks and, instead, focus on specific communication protocols.

For example, CC-Hunter [19] dynamically tracks the cache conflict patterns between processes. It accurately detects covert channel communications that alternate between the trojan sending data and the spy receiving it. However, it fails to detect any variation of the attack that does not involve this strict alternating sequence [7], [8], [6], [10]. Hence, we need more robust detection mechanisms that do not rely on a particular communication protocol and can provide high coverage.

Understanding the different kinds of protocols employed by cache-based covert channel attacks is essential to designing a

* This work was supported in part by NSF under grants CCF-1012759, CNS-1116237, and CCF-1536795.

good detection scheme. Therefore, this paper first develops a new taxonomy of cache-based covert channel attacks.

Based on this taxonomy, we make the following observation about these attacks: not only are the malicious accesses highly tuned to the mapping of addresses to the caches; they also follow a distinctive cadence as bits are being passed between trojan and spy. Changing the mapping of addresses to the caches substantially disrupts the conflict miss patterns, but retains the cadence. This is in contrast to benign programs.

Based on this observation, we propose *ReplayConfusion*, a novel, high-coverage approach to detect cache-based covert channel attacks. *ReplayConfusion* relies on the Record and deterministic Replay (RnR) primitive, where a program's execution is recorded and then replayed deterministically. In *ReplayConfusion*, the replay execution uses a different mapping of addresses to the caches. Then, we analyze the *difference* between the cache miss rate timelines of recording and replay. If the difference function is both sizable and retains a periodic pattern, it indicates that there is an attack.

We find that *ReplayConfusion* identifies example attacks from all the categories in our taxonomy. In addition, it correctly recognizes benign programs. Finally, the hardware needed for *ReplayConfusion* is simple and non-intrusive.

II. Background

A. Cache-based Covert Channel Attack

A covert channel attack occurs when a malicious agent (i.e., a trojan process) periodically leaks sensitive information in a victim system, that a second, malicious process is able to read. In this scenario, the trojan is also called the sender, while the listening spy process is the receiver. In cache-based covert channel attacks, the information that the trojan sends to the spy is encoded in the timing and/or the number of conflict misses in a level of cache shared by both trojan and spy. Most cache-based covert channel attacks target the Last Level Cache (LLC), as it provides the largest attack surface.

A general technique for cache-based covert channel attack is *Prime+Probe* [20], [6], [21]. *Prime+Probe* allows the receiver to figure out which cache sets the sender has accessed. It works as follows. During prime, the receiver fills selected cache sets with its own data. Then it turns idle, and the sender either evicts the receiver's data or leaves it in the cache — depending on the message it wishes to send. Later, the receiver conducts a probe by accessing the same addresses accessed during prime. The receiver learns the sender's decision by estimating the number of misses to these addresses, e.g., with a timer.

Figure 1 shows an example of covert channel attack using *Prime+Probe*. Each square box represents one cache set, labeled with its set ID. The sender and receiver agree on dividing the cache into two groups: even and odd sets. In the prime phase at time t1, the receiver fills all sets. At time t2, the sender evicts the cache lines in the even sets, sending bit "0". In the probe phase at time t3, the receiver accesses both sets, and decodes the information by measuring the difference of timing in accessing even and odd sets. Since it observes a longer delay in even sets, it decodes a "0". The probe process also works as prime for subsequent communication. At time t4, the sender sends bit "1" and at t5 the receiver decodes it.

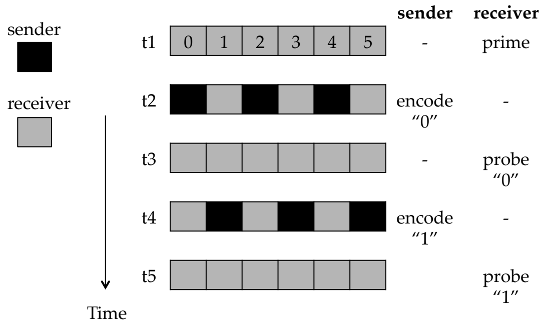


Fig. 1. Example of *Prime+Probe*.

B. Relevant Aspects of Modern LLC Design

Caches are divided into cache blocks, and organized into sets and ways. For performance reasons, the LLC is often further subdivided into slices (also called modules or banks) — each identically configured. To access the LLC, the hardware translates an address into a slice ID, a set index, and a tag. A conflict can only happen between addresses that are mapped to the same slice and set.

The translation process in modern LLCs introduces address uncertainty in attackers in two ways. First, because the LLC is generally physically indexed and physically tagged, the attacker does not have full control over the set where an address maps. Second, the function that selects the slice for a given address (i.e., the Slice Hash function) is undocumented, which makes it difficult for the attacker to figure out the slice where an address maps.

Several techniques have been proposed to figure out the specific set where an address maps. First, large pages (e.g., 2MB) are easier to handle than smaller ones. This is because, with larger pages, more virtual address bits are unchanged by the translation. Figure 2 shows the breakdown of a physical address for a 2MB 16-way cache with 64B blocks. Bits 6-16 are used as the set index. Below that, we have the virtual address when using 2MB pages. All the bits that decide the set index (shown in dark) are within the page offset bits — putting them under attacker control. At the bottom, we have the virtual address when using 4KB pages. Only the lower 6 bits of the index can be controlled by attacker.

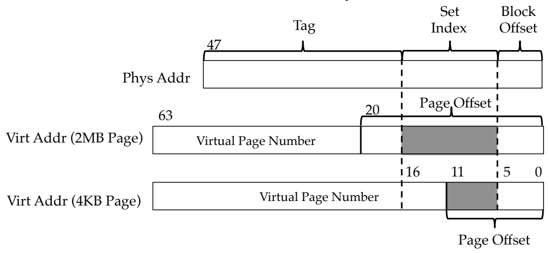


Fig. 2. Address translation example.

Secondly, recent research has shown how to reverse engineer the Slice Hash function of a machine, or construct conflict sets without the need to know the exact Slice Hash function implemented by the hardware. For example, Maurice et al. [22] propose an automatic algorithm that reverseengineers the Slice Hash function. It leverages common per-slice performance counters to detect misses to particular slices. Since there is variation in distance between cores and slices, other work has shown that the slice ID can be determined by detecting access time variations when accessing cached addresses [23]. Both approaches need to access a specific system file to determine the physical frame number of the tested addresses. Other approaches propose techniques to construct precise address conflict sets [24], [10] without knowing the Slice Hash function. All approaches need to read either performance counter data or timer difference.

C. Detecting Cache-based Covert Channel Attacks

Cache-based covert channel attacks are popular because they exploit an attack surface that both is very accessible and provides high bandwidth. Unfortunately, there is no general, low-overhead detection solution. Some of the existing detection techniques apply statistical methods or pattern recognition to find certain features that can differentiate an attack from a legitimate program. Performance counters are often used. For example, Chiappetta et al. [25] detect a covert channel based on the correlation of L3 cache accesses between the trojan and the spy process. Hexpads [26] detects a covert channel by its high cache misses. Both works only find naive attacks with obviously abnormal behaviors.

CC-Hunter [19] is a scheme that detects an attack by studying the pattern of conflicts in a shared cache. It augments the LLC with a "conflict miss tracker", which generates a trace of cache eviction events during program execution. Each event is characterized by a conflict direction. For example the event $C_s \rightarrow C_r$ means "core C_s caused an eviction of data inserted by core C_r". In covert channel communications with alternating sender and receiver accesses to the covert channel, we will discover a recurring pattern of $S \rightarrow R$ events followed by a similar number of $R \rightarrow S$ events. However, as we show later, CC-Hunter cannot detect common attacks where sender and receiver do not strictly alternate accesses to the channel.

D. Record and Deterministic Replay in Security

Record and deterministic Replay (RnR) is a technique where, as a program runs, the system automatically records all non-deterministic events in a log. Then, the program is re-executed, feeding all the non-deterministic events in the log to the program at the appropriate time, creating a deterministic replay [27], [28], [29], [30], [31], [32]. The typical non-deterministic events are inputs to the program (e.g., system calls or interrupts) and the order of main memory access interleaving (for multithreaded programs). The former are typically logged by priviledged software; the latter by hardware.

Prior research has explored RnR for security [33], [34], [35], [36], [37]: to offload the costs of security checks, to analyze intrusions or, generally, to improve program security. Time Deterministic Replay [38] uses RnR to detect timing covert channels in the network. The idea is to compare the timing characteristics of the network packets observed during execution, to the ones observed during the execution of trusted applications. Significant deviations between the two observations suggest that the recorded application was attacked with a covert channel.

III. Threat Model

We assume a strong adversary capable of running a trojan and a spy process on the same machine. The typical attack scenario is shown in Figure 3. An application has a trojan with access to security-sensitive information such as passwords, databases, or security keys. The trojan and the spy process do not share memory thanks to the isolation provided by the operating system (OS) or hypervisor. Hence, no secrets can be leaked through traditional inter-process communication (files, shared-memory or pipes). As such, the trojan is limited to indirectly leaking the information through covert channels.

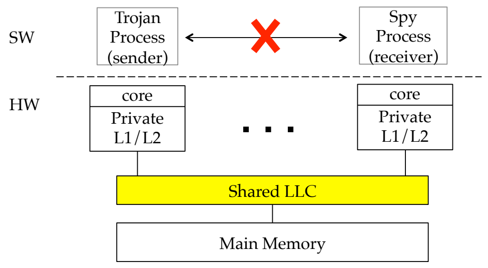


Fig. 3. LLC-based covert channel attack scenario.

Although we describe the sender and receiver as being processes, our scenario extends to virtual machines. In that scenario, the sender and receiver reside in different virtual machines that are scheduled on cores sharing the LLC.

We assume that the OS or hypervisor can secure the record log against corruption by the adversary. Further, we assume that the attacker has full knowledge of the machine's cache configuration and the mapping of addresses to the caches.

We focus on detection of LLC-based covert channels because the LLC is very accessible to attackers, as it is shared among all the cores in a multi-core chip. It is accessible even by adversaries conducting attacks in virtualized environments. Moreover, there are no general, low-overhead defensive solutions for LLC-based covert channels. For example, a technique such as cache partitioning suffers serious performance overhead when supporting a large number of partitions. We present more details in Section IX.

IV. A Taxonomy of Attacks

In cache-based covert channel attacks, a trojan and a spy agree on a protocol to pass information through cache misses, and then the spy uses timing or performance counter measurements to estimate where and when the misses occur. In this section, we introduce a new taxonomy to categorize and understand the design of these protocols. The taxonomy includes all the covert channel communications through LLC misses that we know of.

Figure 4 overviews the taxonomy. It revolves around time and space. Next, we describe these two characteristics.

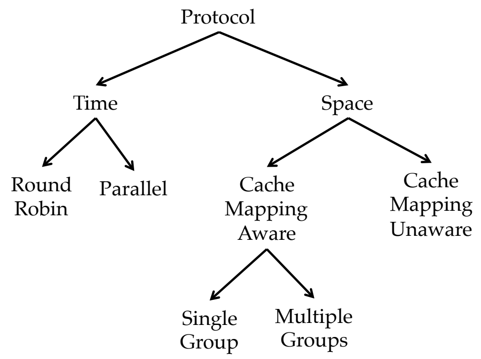


Fig. 4. Taxonomy of cache-based covert channel communication protocols.

A. Time Aspect

The time aspect of the taxonomy classifies the protocols based on the ordering requirements between the sender and receiver interactions with the channel. There are two approaches, based on whether or not there are concurrent accesses of the sender and receiver: round robin and parallel.

A *Round-robin* protocol requires a strict ordering between the sender and receiver, disallowing any concurrent accesses (Figure 5(a)). The sender sends one bit every T_s time units. This window of time is larger than the time it takes for the sender to encode the bit. The remaining time is needed to ensure that there is sufficient time for the receiver to be scheduled and decode the bit. Likewise, the time between successive reads by the receiver (T_r time units) must also be longer than the time it takes for the receiver to decode the bit.

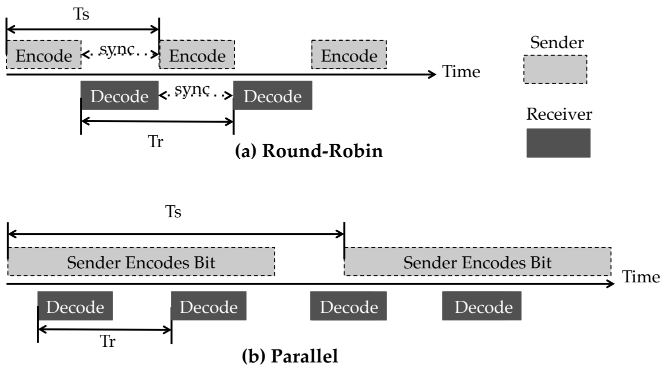


Fig. 5. Classification of the protocols based on the time aspect.

Correctly setting these timings makes the round-robin protocol the most challenging to implement — especially in virtualized environments [8]. Moreover, due to its strict ordering of alternating sender and receiver pattern, the round-robin protocol is easier to detect.

A *Parallel* protocol removes the ordering restrictions and allows concurrent accesses from sender and receiver (Figure 5(b)). This protocol has been proposed to avoid the need to carefully interleave sender and receiver operation [8]. However, the concurrent accesses sometimes introduce errors, which can be overcome by transmitting and decoding a bit multiple times. Hence, the encoding time period is intentionally extended to guarantee that the receiver gets several correct measurements of the bit. Due to the repeated operations to transmit and receive one bit, this protocol takes longer to transmit information.

B. Space Aspect

The space aspect of the taxonomy classifies the protocols based on the addresses that sender and receiver choose to create conflicts. We classify the protocols based on two axes: *Group Count* and *Mapping Strategy*.

1) Group Count: A group is a set of cache sets. The state of group can be either *flushed* or *cached*. To put a group in the flushed state, the sender evicts all the receiver cache lines that were previously cached in it. To put the group in the cached state, the sender issues no accesses to the group, thus leaving all the existing receiver cache lines in it.

A protocol can use a *Single Group* or *Multiple Groups* for the communication between sender and receiver. If it uses a single group, the receiver interprets the bit based on the two possible states of the group. However, when observing the cached state, the receiver is unable to distinguish whether a new bit has already been sent or not. This is a significant drawback, as it leads to bit error rates.

A protocol with multiple groups can use techniques like differential coding to improve reliability. Differential coding represents a bit with two groups that have different values [5]. If the two groups have the same value, then the bit has not been sent yet, or the bit has been corrupted by noise in the system. When the two groups have different values, the receiver can decode the bit based on which group takes longer time to access. The additional groups can also be used to implement multi-bit transmission to improve bandwidth. Note that, to be able to define multiple groups, the protocol must understand how addresses map to the caches.

2) Mapping Strategy: As discussed in Section II-B, the attacker is most effective when he knows the function that the hardware uses to map addresses to the caches. In this case, the attacker uses what we categorize as a *Cache Mapping Aware* protocol. He can create one or more groups, and achieve controllable cache conflicts between the sender and receiver.

Alternatively, if the attacker does not know how the hardware maps addresses to the caches, it uses what we call a *Cache Mapping Unaware* protocol. In this case, he simply allocates a buffer large enough to achieve measurable conflicts. He cannot use multiple groups because he does not know how the addresses map to the caches. Moreover, since there can be a large number of conflicts with the addresses in the buffer, this protocol is easily affected by system noise. Finally, because of the large number of addresses accessed, it takes longer to encode and decode information.

C. Categorizing Existing Attacks

Table I summarizes existing attacks in the literature according to our taxonomy. The protocols in [5], [9] use round-robin protocols and two groups to communicate. Their two-group setup is such that one group contains the odd sets of the LLC while the other contains the even sets. A parallel protocol communicating multiple bits concurrently is demonstrated in [6]. The protocol used in [10] is a parallel protocol that uses techniques to infer the machine's Slice Hash function to minimize the number of accesses required. Two single-group protocols [8], [11] use a fraction of the LLC by choosing addresses based on what set the map to. A parallel, single-group mapping-unaware protocol [7] is shown to be successful when the sender time period is much longer than the receiver's. We have found no examples in the literature of a single-group mapping-unaware round-robin protocol.

	Round robin	Parallel
Multi-group, mapping-aware	[5], [9]	[6], [10]
Single-group, mapping-aware	[11]	[8]
Single-group, mapping-unaware	—	[7]

TABLE I
CATEGORIZING EXISTING ATTACKS.

D. Problems with Existing Solutions

There is currently no general detector of cache-based covert channel attacks. It would seem that an effective detector might try to look for a recurrent pattern of spikes in the cache miss rate during program execution [26]. Unfortunately, this approach is ineffective because such miss patterns are also common in benign programs. This is especially true for programs that have alternating memory-intensive and computation-intensive phases.

Other heuristic-based detection techniques that look for certain memory-access patterns or for frequent time-stamp counter invocations can typically be circumvented once the attacker learns the detection threshold. For example, the attacker can randomize the order of address accesses or lower the time-stamp counter invocation frequency.

As discussed in Section II-C, CC-Hunter is designed to detect strict alternating patterns of cache conflicts between sender and receiver. This pattern occurs in round-robin protocols. However, parallel protocols allow concurrent cache accesses from the sender and the receiver. This eliminates the alternating pattern of conflicts that CC-Hunter detects. We verified that parallel protocol attacks evade the detection by CC-Hunter.

V. REPLAYCONFUSION

A. Main Idea

Our goal is to design a high-coverage detector of cache-based covert channel attacks. To this end, we note two characteristics of these attacks. The first one is that they are carefully crafted to rely on the specific mapping of addresses to cache locations in the machine. Hence, if we change such a mapping, we are very likely to significantly change the conflict miss behavior and disrupt the attack. The second characteristic of these attacks is that the cache conflicts follow a distinctive pattern that broadly repeats over time, as bits are being received by the spy. The exact pattern depends on the actual protocol used by the attack, and the actual sequence of bits received. For example, in some protocols, the repeating pattern is due to probe periods separated by idle periods; in others, it is due to the arrival of the sequence of bits. Hence, after we change the mapping and get a different cache conflict behavior, such a repeating pattern should still be distinguishable.

Based on these insights, we propose *ReplayConfusion*, a novel, general technique that uses Record and deterministic Replay (RnR) to detect cache-based covert channel attacks. While an application executes, RnR support is used to log the non-deterministic events for later replay; in addition, the cache miss rate timeline of the application is also recorded. Then, the application is deterministically replayed (either on-line or off-line) using the log while applying a *different* mapping of addresses to cache locations; the new cache miss rate timeline is also recorded. Finally, we compute the *difference* between the two cache miss rate timelines, and analyze the resulting miss rate difference timeline for a sizable, periodic pattern. The presence of such a pattern uncovers a cache-based covert channel attack. The lack of such a pattern indicates a benign program.

B. Overall Architecture

The shaded boxes of Figure 6 show the components of the *ReplayConfusion* architecture. They include one hardware module (the Cache Address Computation Unit) and three modules in the OS (the Cache Configuration Manager, the Cache Profile Manager, and the RnR Module). The other boxes are present in conventional systems.

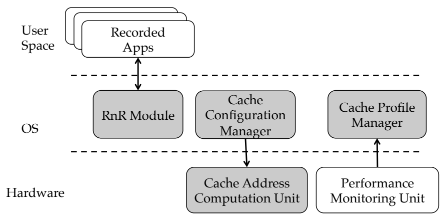

User Space
Recorded Apps
OS
RnR Module
Cache Configuration Manager
Cache Profile Manager
Hardware
Cache Address Computation Unit
Performance Monitoring Unit

Fig. 6. High-level *ReplayConfusion* architecture.

The Cache Address Computation Unit maps physical addresses to the cache hierarchy. It is composed of two hardware functions: the function that picks the cache set for a given address (Set Index Function or f_{set}) and the one that picks the slice for the address (Slice Hash Function or f_{sli}). We call the combination of f_{set} and f_{sli} the Cache Mapping Function (*F*). In modern hardware, *F* is fixed.

In *ReplayConfusion*, we propose to make the two functions in *F* programmable. The Cache Configuration Manager is an OS module that can reconfigure them. Specifically, during the initial (recording) execution, the hardware uses a default Cache Mapping Function F_{def}. During the replay, the Cache Configuration Manager changes the Cache Mapping Function to a new one F_{new}. The F_{new} is different for each process.

The Cache Profile Manager periodically obtains information about the execution, such as the number of instructions executed, LLC accesses performed, and LLC misses suffered. It generates an LLC miss rate trace and securely saves it to a storage system for later processing. All the required information can be obtained from a Performance Monitoring Unit such as the one present in modern processors.

The RnR Module in the OS records all the application's non-deterministic inputs, including scheduling information. Examples of such module have been described in the past [28]. Since this paper is not concerned with multithreaded applications, there is no need for special hardware to record the order of memory access interleaving. More details on the RnR Module are discussed in Section VI-C.

C. Motivating Example

To see the impact of changing F_{def}, consider a protocol with two groups, $G0$ and $G1$, composed of lines mapped to the even and odd cache sets, respectively. The receiver interprets the logic value "1" when it suffers few cache misses on $G0$ and many on $G1$, and logic value "0" in the opposite case. Figure 7(a) shows two time steps during recording: in step *T0*, assume that the sender sends value "1", evicting the lines in group $G1$; in step *T1*, the receiver reads the "1".

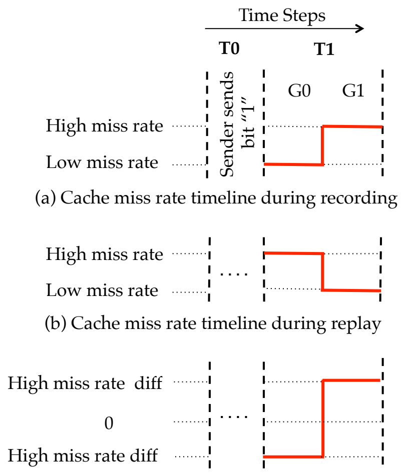


Fig. 7. Effect of changing the Cache Mapping Function.

Assume that, in the replay execution, the sender gets an F_{new} similar to F_{def}, but the receiver gets an F_{new} that, relative to F_{def}, flips the last bit of the Set Index Function. For the receiver, therefore, blocks that used to map to even cache sets now map to odd ones, and vice-versa. The resulting execution is shown in Figure 7(b). In *T0*, the sender still sends value "1". In *T1*, the receiver thinks it is accessing $G0$ and $G1$ but it is in fact accessing $G1$ and $G0$, hence observing the opposite miss rate pattern.

Figure 7(c) shows the difference between the cache miss rate during recording and the cache miss rate during replay. Our small hardware change has caused a significant change in the miss rates.

D. Designing F_{new}

We try to design F_{new} so that it has a minimal effect on most benign programs while causing significant changes to the cache behavior of malicious programs. In the following, we discuss the Set Index Function (f_{set}) component and the Slice Hash Function (f_{sli}) component in turn.

1) Set Index Function: The default Set Index Function in modern hardware is:

$$index = \frac{phys\_addr}{block\_size} \% set\_count$$

We propose to take the computed $index$ bits and either flip bits or swap bits. For example, we can flip the last bit of $index$ like in the example of Section V-C, or swap the first and last $index$ bits. With these changes, the number of self-conflict misses within a program remains the same. The same is true for the conflict misses between all the processes in the machine — provided that we use the same f_{set} for all the processes. However, if different f_{set} functions are used for different processes, the cache interference between the processes will be different.

One insight of our work is that benign programs are written to be oblivious to the existence of other programs in the machine. Hence, using different f_{set} functions for different programs will not generally affect their inter-process cache conflict misses much. However, a trojan and a spy process generate inter-process cache conflicts intentionally, using the fact that both processes use the same f_{set}. If, during replay, we use different f_{set} functions for the trojan and for the spy, the number of inter-process cache conflict misses will change substantially.

2) Slice Hash Function: The Slice Hash Function maps physical addresses to LLC slices (Section II-B). The default function in modern systems is an *xor* function on selected bits of the physical address. We propose to change the address bits used by the hash function during replay. However, we must be careful so that addresses are still largely uniformly distributed across slices, and the cache behavior of benign programs is impacted little. One of the ways to encourage this is by only changing the original f_{sli} slightly. For example, the new function can replace a few bits with adjacent ones.

Since benign programs are oblivious to f_{sli}, such f_{sli} modifications will typically not introduce significant cache miss rate changes to benign programs. However, an attacker may leverage knowledge of f_{sli} in order to define groups in particular slices. It is for these kinds of attacks that changing f_{sli} is effective. By changing the mapping of addresses to slices between trojan and spy process, many conflicts between the two will likely be eliminated, causing large deviations from the miss rate of the recorded execution.

E. Overall Operation

Figure 8 shows the operation of *ReplayConfusion*. The Cache Address Computation Unit has several Cache Mapping Functions (F_i) that can be selected and applied (F_{sel}) to a memory address to generate its cache mapping. Initially, the Cache Configuration Manager configures the Cache Address Computation Unit to use F_{def} ①. During the initial (recording) execution of a set of programs, the RnR Module records in a log all the non-deterministic events in the execution ②, while the Cache Profile Manager records the cache miss rate timeline of the execution ②.

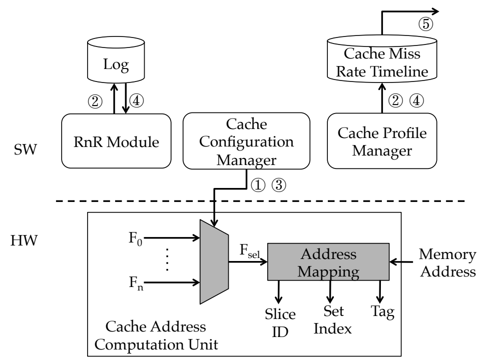


Fig. 8. Operation of *ReplayConfusion*.

Concurrently or later, the log is transferred to a different machine and the processes are replayed to check for possible cache-based covert channel attacks. During the replay, the Cache Configuration Manager configures the Cache Address Computation Unit to use a different F_{new} ③ for each process as it is scheduled. This F_{new} is associated with that process and all its children threads. The RnR Module reads the log and steers the replay to be deterministic ④, while the Cache Profile Manager records the cache miss rate timeline of the replay ④.

Also concurrently or later, *ReplayConfusion* subtracts the replay cache miss rate timeline from the recording cache miss rate timeline. The result is the miss rate *difference timeline* (T_{diff}). It then analyzes it to detect whether there is a cache-based covert channel communication ⑤.

The process described is flexible. For example, the system administrator can create a rule to check every untrusted process. Alternatively, one can check only the processes with suspicious activity, such as those with frequent execution of timer read instructions or access to a specific system file containing physical address information.

The analysis of T_{diff} is based on the following idea. For a benign program, the modified address mapping is likely to have a negligible impact and, hence, most values in T_{diff} should be close to zero. In the case when they are not close to zero, they should not exhibit a periodic pattern with time. However, for a malicious program, we should see significant values of T_{diff}, as the new mapping disrupts the original pattern of conflict misses. Moreover, we should see a broadly repeating pattern with time, as the spy repeatedly reads information from the channel.

ReplayConfusion looks for repeating patterns in the T_{diff} signal. To detect these patterns, it generates the autocorrelation of the T_{diff} signal. In signal processing, autocorrelation is a technique used to identify periodic patterns in a signal [39]. Autocorrelation computes the similarity of a signal with itself, when shifted by a varying number of time lags (or samples, for the discrete case). The autocorrelation of a signal X at a particular time lag p is:

$$C_p = \frac{n \sum_{i=0}^{n-p} [(X_i - \overline{X})(X_{i+p} - \overline{X})]}{(n-p) \sum_{i=0}^{n} (X_i - \overline{X})^2}$$

where X_i is the value of the signal X at time i, X_{i+p} is the value of X at the instant that is p samples after X_i, $\overline{X}$ is the mean value of X, and n is the total number of discrete samples of X.

The autocorrelation function plots how C_p varies with the time lag p. This measure takes the value of 1 when the signal and the variant shifted in time by p samples perfectly align with each other — i.e., all values of X_i and X_{i+p} match exactly. The measure takes a value of -1 when the signal and its shifted variant are perfectly oppositely aligned — i.e., the peaks of X_i occur where the valleys of X_{i+p} occur. When the signal has no periodic patterns, the value is close to 0.

When the signal is periodic with period T, there is a peak in the autocorrelation for a time lag equal to the period of the signal (i.e., C_p is 1 for $p = T$), since the signal shifted by a period is equal to the signal itself. Additionally, there are peaks at lags of $2 \times T$, $3 \times T$, and so on, since the signal shifted with these lags is also equal to the signal. For lags that are around the middle of a period, such as $0.5 \times T$, the autocorrelation has a valley. For other lags, the values are in-between the two extremes. Therefore, the autocorrelation of a periodic signal shows consecutive values of maximum and minimum extremes.

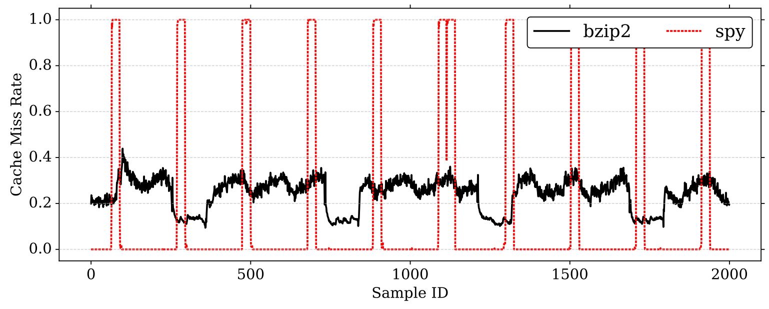
The plot of the autocorrelation function is called autocorrelogram. *ReplayConfusion* computes the autocorrelogram of the T_{diff} signal of a program. If it finds consecutive peaks and valleys, it classifies the program as an attack.

F. Example

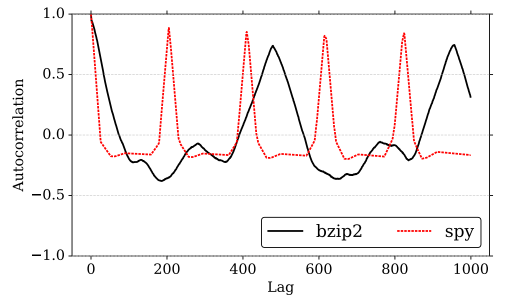
Figure 9 shows an example of how *ReplayConfusion* distinguishes a malicious program from a benign one. Our example benign program is bzip2 from SPEC2006 (running together with h264ref). Our example malicious program is the spy process in a trojan and spy process pair that uses a parallel protocol with two groups and cache mapping awareness. Each group has 1MB of data. Recall that, in this protocol, during the probe period, the spy process reads one group that misses in the LLC and one group that hits. The order depends on the actual logic value of the bit encoded in the transfer.

Figure 9(a) shows the cache miss rate timeline of bzip2 and the spy process as a function of time measured in samples. Each sample is 100,000 cycles of recording execution (more details are given in Section VII). We see that bzip2 has a pattern of high and low miss rates. Such a pattern is common in programs that contain alternating memory-intensive and computation-intensive phases. The spy process also has a repeating pattern. The repeating pattern includes two periods: one with high miss rates and one with low miss rates. The former corresponds to the spy process' probe period when it reads the group that misses in the LLC. It covers several samples, which have a miss rate of 1.0. The period with low miss rates corresponds to the spy process' probe period when it reads the group that hits in the LLC and, also, an idle or waiting period, when it is performing useless work or spinning in a loop, while avoiding causing misses in the LLC.

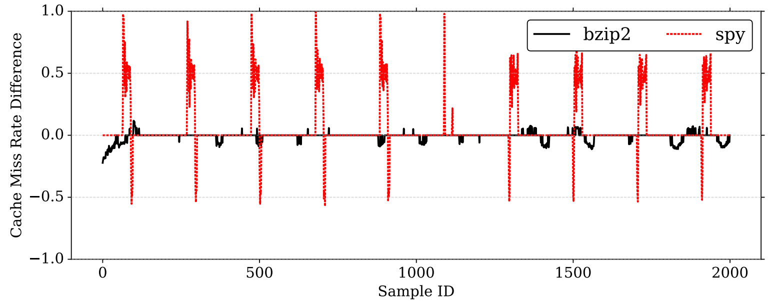
Figure 9(b) shows the autocorrelogram of the cache miss rates for bzip2 and for the spy process. We can see an



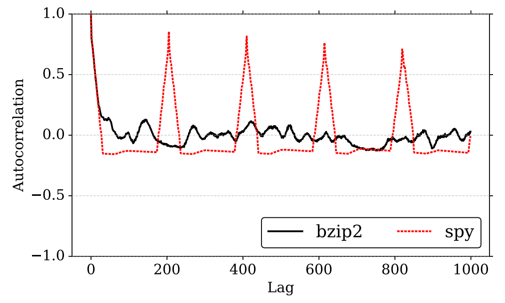
(a) Cache miss rate timeline



(b) Cache miss rate autocorrelogram



(c) Cache miss rate difference timeline



(d) Cache miss rate difference autocorrelogram

Fig. 9. Comparing a benign and a malicious program.

oscillatory pattern in both autocorrelograms. Therefore, cache miss rate alone is not enough to be used to distinguish benign from malicious programs.

Figure 9(c) shows, for each of the two programs, the cache miss rate difference between the recording execution and the replay execution (T_{diff}). In the replay, we have changed both the Set Index and the Slice Hash Functions. Details on how the recording and replay samples are aligned, and how we have changed the functions are presented in Section VII. We see that, for bzip2, most of the samples have a very low cache miss rate difference (*CMD*). This means that our mapping changes do not affect the cache miss rate timeline much. For the spy program, however, we get significant values of the *CMD* during the probe period. This is because, due to address remapping, the group that was supposed to cause only misses now also exhibits some hits, and the one that was supposed to cause only hits now also has some misses. The miss difference can be positive or negative. The idle period has a low miss rate in both record and replay, so the *CMD* is negligible.

Finally, Figure 9(d) shows the autocorrelation of the absolute value of cache miss rate difference. In the figure, we see that bzip2 does not show any sizable pattern. However, for the spy process, we see a repeating pattern. This pattern is created by the repeating pattern of high *CMD* during the probe period and low *CMD* during the idle period. The peaks in the autocorrelogram appear at time lags that are multiples of the time between probe periods. In the example, the time between probe periods is about 200 samples. Hence, the peaks give us the bandwidth of the communication. Note that the peaks are not perfect and do not reach 1.0 because of noise.

In summary, *ReplayConfusion* detects this attack because trojan and spy communicate with a pattern of conflict misses that, after subtracting the miss rate during replay with a modified address mapping, results in a miss rate difference that (i) is significant and (ii) still preserves a broadly periodic pattern. The autocorrelogram exposes this pattern. We will analyze multiple attacks in Section VIII.

G. Detection Effectiveness

The effectiveness of *ReplayConfusion* depends on its ability to distinguish attacks from benign programs by their different sensitivity to address mapping changes. In our experiments, we have not found cases where *ReplayConfusion* suffers false positives or false negatives. This does not mean that they do not exist; they may be uncommon. In this section, we first describe why false positives should be rare, and then discuss several examples of advanced attacks and why they do not lead to false negatives.

1) False Positives Are Rare: False positives should be rare because, for a benign program to be classified as a spy, it needs to satisfy two conditions. First, it should exhibit very different cache miss rates under the original mapping function and under the new one. Second, the difference between the two miss rates should have a repeating pattern. Recall from Section V-D that the F_{new} is selected so that the self-interference misses within a process barely change. Hence, any miss rate change should be due to interference with another program. Moreover, the execution of that program and the benign one should appear to be synchronized in a way that their cache interference patterns keep repeating. This is very unlikely. Still, if it happens, one would give the binary to a security expert for analysis.

2) Advanced Attacks that Do Not Lead to False Negatives: In one advanced attack, the trojan and the spy process choose the conflict sets dynamically. Specifically, in an initial phase before the actual communication, the trojan continuously accesses a group of cache sets. The spy reads many sets and, using timers or performance counters, finds the group of sets that frequently miss in the cache. *ReplayConfusion* foils this attack because, during the original execution, the RnR Module sees these requests to the timer or performance counters and logs the responses (since they are non-deterministic inputs to the program). During the replay execution, the RnR Module feeds these same values to the spy's timer or performance counter requests, effectively providing incorrect information, and causing the trojan and spy to choose the same conflict sets as during recording.

Another advanced attack involves using low communication bandwidth — i.e., sending information only over a long time. *ReplayConfusion* can detect the attack by monitoring over a longer period of time. In this case, the autocorrelogram will still show an oscillatory pattern, with a larger gap between peaks.

Other attacks use few cache misses to communicate between the trojan and the spy process. To detect these cases, *ReplayConfusion* needs to perform more frequent sampling of events (e.g., LLC misses, LLC accesses, and instructions executed). Otherwise, the extra misses involved in the information transfer would be overlooked. We evaluate this case in Section VIII-D.

There are attacks that introduce noise, or mimic benign programs. These attacks are hard to implement, because the noise should be random, and sizable enough to hide the attacker's behavior. However, such noise also hurts the attacker's communication reliability.

Finally, one true false negative can occur if the two new mapping functions assigned to the trojan and to the spy process (F_{new_tr} and F_{new_sp}), together create similar conflict misses in the spy as the old mapping function F_{def} assigned to both processes did. This is very unlikely because there is a very large design space of F_{new}.

VI. Implementation Details

A. Flexibility of the Cache Address Computation Unit

The Cache Address Computation Unit is the only hardware unit modified by *ReplayConfusion*. As discussed in Section V-B, this unit is modified to allow the software to change the address mapping.

To generate the Set Index Function, we add a special register with information on all of the bits in the cache index field. For each of these bits, the register specifies if the bit is inverted and/or if it is swapped with what other bit. To generate the Slice Hash Function, we add a special register with information on which bits are XOR'ed to participate in the generation of the slice ID.

These two registers can only be configured by the Cache Configuration Manager. Also, when a process is scheduled during the replay, the Cache Configuration Manager ensures that the correct F_{new} is applied. These simple modifications have negligible performance impact on the system.

B. Program Execution Correctness

The OS must ensure that the program executes correctly at all times, even though the mapping of addresses to the cache hierarchy changes dynamically. First, threads that share memory must use the same mapping function to correctly access the same cache line. Hence, during the record phase, the OS keeps track of the shared-memory activities, and tags the threads that share memory with the same tag. During replay, all threads that have the same tag are assigned the same F_{new}.

We also need to handle cache line writebacks carefully. When a dirty cache line is evicted from a cache, its correct physical address needs to be computed to write the line back to the next cache level or main memory. In conventional systems, the physical address is computed from the tag, set index and block offset. However, with *ReplayConfusion*, we need to use the correct F mapping function.

Recall that, during replay, each core may be using a different F_{new} (assuming no SMT), and F_{new} may change at every context switch. Consequently, we modify the hardware so that, every time that a cache line is updated, it is tagged with the ID of the writing core. Moreover, the cache controller of each slice of the shared LLC contains the F_{new} of all the processes that are currently running in any of the cores. In addition, the cache controller of each private cache contains the F_{new} of the process that is currently running locally.

With this support, every time that a dirty line is evicted from a cache, the cache controller uses the line's tag to select the correct F_{new}. It then uses that F_{new} to generate the correct physical address and write the line to the correct location in the next cache level or memory.

In addition, when core C context switches, the cache controllers of its private caches and of the shared LLC slices go over all the cached lines and write back all the dirty lines with tag C (using the correct F_{new}). This adds some performance overhead during context switching, but it only happens during replay, off the critical path. The OS instructions and data are unaffected because they always use F_{def}.

C. RnR Operation

Most application-level RnR schemes only support functional replay (e.g., [34], [27], [40], [41], [33]). This means that, across threads, they only reproduce the relative order of events that are functionally dependent — e.g., through a data dependence. In *ReplayConfusion*, we additionally need to preserve the relative order of memory system accesses across threads. This would seem to be hard. Note, however, that we do not necessary need to preserve the order of memory accesses at the granularity of individual accesses.

As indicated in Section V-B, as an application is being recorded, the Cache Profile Manager periodically takes measurements every fixed number of cycles. It records information about the execution, such as the number of instructions executed, and the number of LLC accesses and misses. *ReplayConfusion* uses this information to ensure that the recording

F_{def}

f_{set}	$(pa/64)\%1024$
f_{sli}	bit$_0$: $p18 \oplus p19 \oplus p21 \oplus p23 \oplus p25 \oplus p27 \oplus p29 \oplus p30 \oplus p31$ bit$_1$:$p17 \oplus p19 \oplus p20 \oplus p21 \oplus p22 \oplus p23 \oplus p24 \oplus p26 \oplus p28 \oplus p29 \oplus p31$

F_{new} for replay

f_{set}	Take f_{set} from F_{def} and swap the most significant and least significant 5 bits
f_{sli}	bit$_0$: $p17 \oplus p19 \oplus p20 \oplus p22 \oplus p24 \oplus p26 \oplus p28 \oplus p30 \oplus p31$ bit$_1$: $p18 \oplus p20 \oplus p21 \oplus p22 \oplus p23 \oplus p24 \oplus p25 \oplus p27 \oplus p29 \oplus p30$

Fig. 10. F_{def} and F_{new} Cache Mapping Functions. In the tables, *pa* stands for physical address, and p_i for bit i of the physical address.

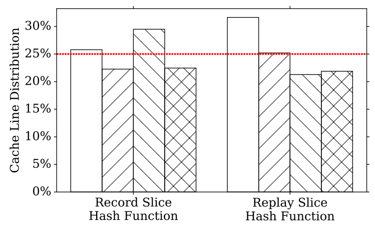


Fig. 11. Cache line distribution across slices.

and replay executions are aligned at least at the granularity of these samples.

Specifically, the number of instructions executed in each of the samples during the recording execution is taken as a reference. During replay, we use instruction count interrupts to interrupt a sample when it has executed the same number of instructions as the corresponding sample during recording. Then, the thread's execution is paused until all the other threads reach the end of their corresponding sample. At that point, all threads are resumed. With this support, the different threads preserve the relative order of memory system accesses at the granularity of samples.

VII. Experimental Setup

We evaluate *ReplayConfusion* using MARSSx86 [42], a full system simulator. We boot Ubuntu 10.4 with a 4GB main memory. Hence, we use 32-bit physical addresses. The simulator models 4 in-order cores, with 32KB private L1 caches, and a 2MB shared L2. The block size is 64B. The L2 is organized in 4 slices. Each slice is 8-way set-associative and hence contains 1024 sets.

In our experiments, we execute the trojan in one core and the spy in another core. In the default, noiseless environment, the other two cores are idle; in the environment with background noise (Section VIII-C), the other two cores run the Twitter application from OLTP-Bench [43]. We also run experiments with benign applications. For these, we use bzip2, h264ref, gobmk, and sjeng from SPEC2006 [44], and the Stream program [45]. The latter is configured to use an array size similar to the LLC size.

Figure 10 shows F_{def} (used in recording) and F_{new} (used in the replay of the spy). Since L2 has 4 slices, f_{sli} is given as two bits, which together denote the slice ID. The f_{sli} in F_{def} uses the Slice Hash Function of the Intel i7-4702M Haswell [24]. The replay f_{sli} is carefully chosen to ensure it balances the address distribution across slices like the record f_{sli}. We verify this by mapping 2M addresses using 512 4K-pages. Figure 11 shows the resulting cache line distribution for both record and replay f_{sli} functions. If a perfect hash function is used, each slice will get exactly 25% of the cache lines. Our record and replay f_{sli} functions show a slight imbalance: individual slices get between 20% to 30% of the lines. This imbalance is acceptable.

During the recording execution, *ReplayConfusion* obtains the number of instructions, LLC accesses, and LLC misses every 100,000 cycles — which we call a *sample*. To perform any meaningful comparison between the recording and replay runs, we need to compare samples that have executed the same instructions. To accomplish this, the samples during replay are terminated when they have executed the same number of instructions as the corresponding samples during recording. To accomplish this, we use interrupts, as explained in Section VI-C.

ReplayConfusion then computes the cache miss rate per sample in the record and replay execution, generating two cache miss rate timelines as a function of the sample number. It then creates another timeline with the absolute value of the difference between the cache miss rates of corresponding record and replay samples. Finally, it computes the autocorrelation of this cache miss rate difference timeline.

VIII. Evaluation

A. Attacks Using Different Protocols

We evaluate the autocorrelograms obtained with attacks using different protocols. We consider both parallel and round-robin versions of the following protocols: (i) two-group mapping-aware (*2g*), (ii) single-group set-index mapping-aware (*1g_set*), (iii) single-group slice-hash mapping-aware (*1g_slice*), and (iv) single-group mapping-unaware (*unaware*). *1g_set* and *1g_slice* differ in that the first one knows f_{set}, while the latter knows f_{sli}. Figures 12 and 13 show the autocorrelograms for the parallel and round-robin protocols, respectively.

We first consider the parallel protocols. In all of them, we set $T_s = 5 \times T_r$, so that the spy gets a given bit 4 or 5 times. Also, we set T_r to be proportional to the number of cache sets used for communication. *2g* is the attack used by Xu et al. [9] and given as example in Section V-F. In this attack, the whole LLC is divided into an odd and an even group based on the last bit of the set index. The sender uses a differential encoding of the two groups to send a bit. When the spy misses in the first group and hits in the second one, it interprets it as a logic "1"; when it hits and then misses, it is a logic "0". In our example, the spy reads "1" five times and then "0" five times (Figure 9(a)).

During the probe phase, no matter what logic bit is transmitted, the Cache Miss rate Difference (*CMD*) between record and replay is high (Figure 9(c)). The *CMD* is positive and then negative, or vice-versa, depending on the actual bit transferred. During the idle phase, however, the *CMD* is close to zero.

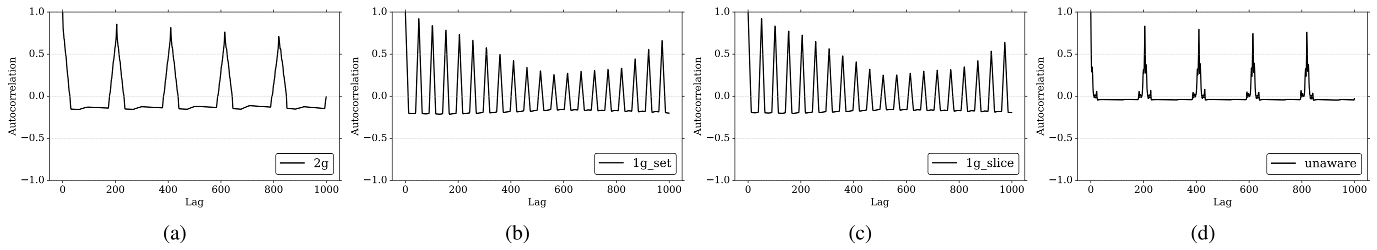


Fig. 12. Autocorrelograms for different attacks using parallel protocols.

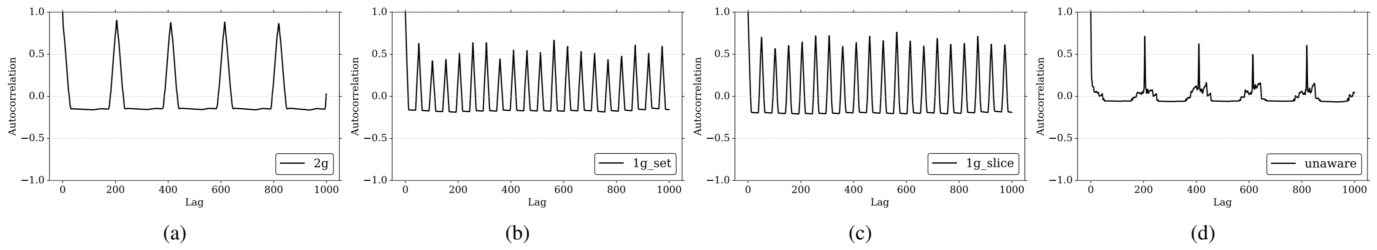


Fig. 13. Autocorrelograms for different attacks using round-robin protocols.

We then compute the timeline of the absolute value of the difference between record and replay miss rates. Finally, we compute the autocorrelation of that timeline. The repeating sequence of probe and idle phases results in an oscillatory pattern in the autocorrelogram (Figure 9(d), which is repeated in Figure 12(a)). Since a pair of probe and idle phases lasts 200 samples, the autocorrelogram shows peaks at every 200-sample lags.

1g_set and *1g_slice* use a group size equal to one quarter of the LLC. The sender sends a value "1" by flushing the whole group, and a value "0" by leaving the receiver's data cached. In *1g_set*, the group contains addresses mapped to the top 256 sets of each of the LLC slices; in *1g_slice*, the group contains addresses mapped to all the 1024 sets of slice 0. In these particular examples, the sender sends bits "1101110".

When receiving bit "1", the spy gets a miss rate close to 1.0 during recording, and about 1/4 of that during replay. This is because the spy now accesses data from the whole LLC — not just from the 1/4th of the LLC that the sender flushed. When receiving bit "0" or in idle phase, the spy exhibits only a low cache miss rate in both recording and replay. This is because the sender has not flushed any part of the LLC.

Since the spy receives each bit 4-5 times, when it receives bit "1", we obtain a repeating high and low *CMD*, corresponding to the probe and idle phase; when it receives bit "0', the *CMD* is always low. Figures 12(b) and 12(c) show the autocorrelogram for the *1g_set* and *1g_slice* attacks. We can see they have peaks, which are caused by the "1" pattern. The lag between each peak is approximately 1/4th as much as in *2g*. This is because the probe-idle phase takes 1/4th as long as in *2g* (because only 1/4th as many sets are accessed). The presence of "0" bits in the message causes the heights of the peaks to vary with the lag.

In *unaware*, sender and receiver communicate through a 2MB buffer, which is the size of the LLC. The sender sends bit "1" by flushing the whole group, and bit "0" by leaving the receiver's data cached. In our particular example, the sender sends bits "00".

In the idle phase, the spy has a low miss rate in both recording and replay. In the probe phase, the miss rate changes between recording and replay because the number of self evictions in the spy changes. Consequently, the *CMD* has a pattern that peaks at every probe period. This causes an oscillatory pattern in the autocorrelogram (Figure 12(d)). The lag between peaks is like in *2g* because both attacks use the same number of addresses in the probe.

We now consider the round-robin versions of the protocols described. We set T_s=T_r and, therefore, each bit is transmitted once. Their autocorrelograms are shown in Figure 13. We see they are similar to those of the parallel protocols. However, since sender and receiver are better aligned in round-robin protocols, the autocorrelograms often show a clearer oscillatory pattern, with higher or less variable peaks. Note that, for the same number of samples, round-robin attacks can transmit more bits than parallel ones.

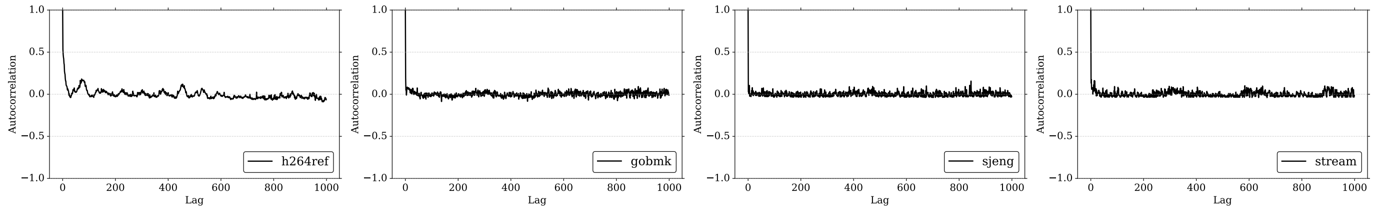
B. *Effects on Benign Applications*

To see the different effect of *ReplayConfusion* on benign applications, we run three workloads of two applications each: bzip2 with h264ref; gobmk with sjeng; and two Stream instances. We run both the recording and the replay executions, and generate autocorrelograms for each of the applications. They are shown in Figures 14(a) for h264ref, 14(b) for gobmk, 14(c) for sjeng, 14(d) for Stream, and 9(d) for bzip2.

We see that the autocorrelograms are flat, in contrast to those of an attack. The reason is that the change in Cache Mapping Function (*F*) between recording and reply does not alter the interference miss rates in a manner that the miss rate difference timeline exhibits any sizable periodic pattern.

C. *Detection with Background Noise*

To evaluate *ReplayConfusion*'s detection effectiveness with background noise, we re-run the *2g* attack with a noise-

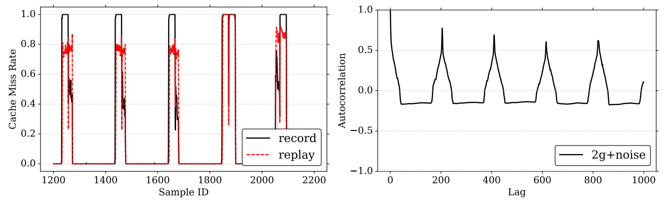


(a) (b) (c) (d)

Fig. 14. Autocorrelogram for benign applications.

generating application running on each of the two otherwise idle cores. The application is Twitter from OLTP-Bench [43], which simulates a database workload. We configure each Twitter application to run with 2 worker threads, each operating at 1,000 transaction per second.

Figure 15(a) shows the cache miss rate profile of the spy program. Consider recording first. As expected, the spy gets a cache miss rate of 1.0 when accessing the group that the trojan has flushed. However, it does not get a miss rate of 0.0 when accessing the group that the trojan has not flushed. Instead, we see from the figure that it gets a miss rate of approximately 0.5. These cache misses are caused by the noise-generating applications. Consider now replay. Without noise, the miss rate during the probe period should be around 0.5 (Section V-F). In this experiment, we see that the miss rate is about 0.8 due to the noise. The result is that, although the *CMD* between record and replay is slightly smaller than the one obtained in the noiseless setup, there is still a repeating pattern. The resulting autocorrelogram still has the expected features, as shown in Figure 15(b).



(a) (b)

Fig. 15. Covert channel attack detection with noise.

When more noise is introduced, more cache lines are evicted by the noise application. With a lot of noise, the spy gets many misses when accessing any of the two groups, leading to invalid measurements. The spy is likely to suffer intolerable error rate before the autocorrelogram cannot detect the attack.

D. Detecting Attacks Using a Small Number of Sets

A commonly-used approach to evade detection is to reduce the group size, namely the number of cache sets used in the attack. However, *ReplayConfusion* can detect attacks with very small group size by reducing the monitoring granularity (i.e., the sample size) appropriately.

In this experiment, we reuse the *2g* attack of Section VIII-A with groups of only 64 cache sets. In addition, the attacker cleverly tries to lower the cache miss rate by issuing a large number of LLC cache hits during the idle phase.

In the idle phase, the miss rate is close to zero in both recording and replay. In the probe phase, the group that has a 1.0 miss rate during recording gets a very low miss rate during replay. This is because the few sets used by the trojan and by the spy are not likely to overlap much. This is the *CMD* that *ReplayConfusion* needs to detect. The group in the probe phase that has a zero miss rate during recording keeps the zero miss rate during replay.

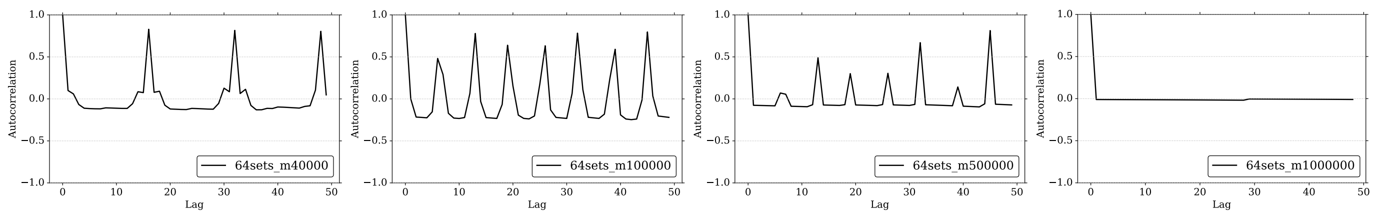
To detect this attack, *ReplayConfusion* should not increase the monitoring granularity. Figure 16 shows autocorrelograms with different monitoring granularities: 40,000, 100,000, 500,000, and 1,000,000 recording cycles per sample. When the granularity is 40,000 or 100,000, each probe and idle phase pair takes multiple samples. Although the *CMD* is low, the repeating pattern is still clear, and the autocorrelograms of Figures 16(a) and (b) show an oscillatory pattern. The lag between peaks in Figure 16(a) is about 2.5 times longer than the one in Figure 16(b). This is because each sample has 2.5 fewer cycles.

With a monitoring granularity of 500,000 or 1,000,000 cycles, multiple probe and idle phases are included within a single sample. This coarser granularity, together with the LLC cache hits introduced by the attacker, end up hiding the repeating pattern. In Figure 16(c), there is still some oscillatory pattern, with lower peaks; in Figure 16(d), there is no oscillatory pattern.

E. ReplayConfusion Overhead

ReplayConfusion uses simple, non-intrusive hardware for its address remapping mechanism. In addition, to support RnR, it needs special software infrastructure (and, if multithreaded programs are to be replayed, some race-recording hardware). The performance overhead that recording adds to a program's execution depends on how efficient the RnR implementation is. The QuickRec hardware prototype [27] found that the average recording overhead is only 13%, and mainly comes from the software stack — i.e., the OS logging all the non-deterministic inputs, such as reads from network or disk, and rdtsc calls.

ReplayConfusion requires replay, but replay is done in a different machine off the critical path. Hence, it does not slow down the program. Replay can be conducted either offline or online. For online replay, the log generated by the execution being recorded is streamed in real time to another machine for concurrent replay. In this way, cache-based covert channel attacks can be detected in real time — albeit with a certain lag. The time-to-detection consists of the time to transfer the



(a) Samples of 40,000 cycles (b) Samples of 100,000 cycles (c) Samples of 500,000 cycles (d) Samples of 1,000,000 cycles

Fig. 16. Autocorrelograms of the *2g* attack using different monitoring granularities.

log entries in real time to the replay machine, where the application is being replayed, compute the miss rate difference between the two executions, and generate the autocorrelogram. In the best case, this process takes in the order of seconds.

Covert channel attack detection is very useful even if it happens after some information has already been leaked. This is because the goal of covert channel attacks is to attain long-term information leakage. As soon as *ReplayConfusion* detects the leak, we can block future information leakage.

IX. Related Work

We discussed CC-Hunter in Section IV-D. Our work is different from CC-Hunter in the following ways. CC-Hunter relies on strict alternating patterns of conflicts between sender and receiver. This only occurs in round-robin protocols. *ReplayConfusion* also detects attacks based on parallel protocols, which allow concurrent accesses by sender and receiver. Moreover, CC-Hunter requires intrusive hardware. It requires a bloom filter and multiple metadata bits per cache line to track conflict misses. *ReplayConfusion* has a lower hardware cost, as it only slightly augments the cache address mapping logic.

Using performance counters to detect side-channel attacks has been proposed. Chiappetta et al. [25] detect side channels based on the correlation of LLC accesses between victim and spy process. Hexpads [26] detects covert channels by their high cache misses. Unfortunately, both works only detect naive attacks with obviously abnormal behaviors. These approaches are ineffective to detect advanced attacks. *ReplayConfusion* relies on the miss rate difference between record and a replay with a different mapping function. It is much harder to bypass.

Chen et al. [38] propose Timing Deterministic Replay for network covert channel detection. They compare the execution of the application with a potential attack (recording) to the execution of a clean implementation of the application (replay). Significant deviations observed in network packet timings indicate a possible attack. *ReplayConfusion* is different in two ways. First, we focus on cache-based covert channels. Their approach cannot be used because timing deviations introduced by cache behavior variations may not be significant enough to be observed by their timing detector. Second, we do not require a clean version of the application, which is not always available as a reference; we record and replay the same binary.

Two classes of techniques have been proposed to prevent cache-based covert channel attacks: cache partitioning and run-time diversification. Cache partitioning prevents interference between programs by assigning exclusive use of part of the cache to each process. Static partitioning [46] often causes cache under-utilization and, as result, significant performance degradation. Stealthmem [12] uses dynamic page coloring to give security-sensitive processes a set of locked cache lines that are never evicted. Catalyst [16] proposes a hardware-software hybrid approach by leveraging the Cache Allocation Technique (CAT) introduced by Intel processors. PLCache [13] supports fine-grained cache locking using special load and store instructions.

These approaches do not work well for cache-based covert channels for two reasons. First, these approaches require identifying security-sensitive programs, or the security-sensitive components within programs. In practice, programs performing a covert-channel attack do not cooperate to provide such information. Second, all current solutions can only support a very limited number of partitions efficiently. For covert-channel defense, we need a large number of partitions to isolate all active processes in the system. This will cause intolerable performance overhead due to frequent cache flushes.

SecDCP [17] is a dynamic cache partition scheme where each application is assigned to a security class. The actual partition decision only depends on the demands of the lower security-sensitive applications. SecDCP can attain low overhead only when concurrently executing applications are in different classes. When multiple applications are in the same security class, the scheme works like a static one.

Under run-time diversification, Wang et al. [13] propose to dynamically randomize the memory line mapping in L1. This prevents the attacker from creating controllable conflicts. However, while this technique works against covert channel attacks that require precise address mapping information, there are address mapping-unaware covert channel attacks, which the technique cannot prevent.

Fuzzytime [15] and Timewarp [14] disrupt the timer by adding noise to reduce the accuracy of the system clock. This general approach cannot prevent covert-channel attacks that tolerate timer noise by using a large number of sets for communication. Besides, it cannot block attacks that use performance counters [11] of cache misses instead of measuring time differences. Furthermore, it hurts benign programs that require a high-precision clock.

Liu et al. [18] propose the Random Fill Cache to defend against reuse-based cache side channel attacks. On a cache miss, instead of filling the cache with the requested data, the requested block bypasses the cache and another cache block in a neighborhood window is loaded into the cache.

This approach is not effective for conflict-based cache covert-channel attacks, where the attacker communicates through a large buffer with many cache conflicts.

X. Conclusion

In this paper, we began by studying the design of cache-based covert channel attacks, and proposed a new taxonomy of these attacks. Using this knowledge, we observed that in these attacks, not only are the malicious accesses highly tuned to the mapping of addresses to the caches; they also follow a distinctive pattern as bits are being received. Changing the mapping of addresses to the caches substantially disrupts the conflict misses, but retains the above pattern. This is in contrast to benign programs.

Based on this observation, we proposed *ReplayConfusion*, a novel, high-coverage approach to detect cache-based covert channel attacks. After a program's execution is recorded, it is deterministically replayed using a different mapping of addresses to the caches. We then analyze the difference between the cache miss rate timelines of the two runs. If the difference function is both sizable and exhibits a periodic pattern, it indicates that there is an attack. We found that *ReplayConfusion* identifies example attacks from all the categories in our taxonomy. In addition, the hardware needed for *ReplayConfusion* is simple and non-intrusive.

References

[1] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, 1973.

[2] S. Cabuk, C. E. Brodley, and C. Shields, "IP covert timing channels: Design and detection," in *CCS*, 2004.

[3] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia, "Model-based covert timing channels: Automated modeling and evasion," in *RAID*, 2008.

[4] G. Shah, A. Molina, M. Blaze *et al.*, "Keyboards and covert channels," in *Usenix security*, 2006.

[5] 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 *CCS*, 2009.

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

[7] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: cross-cores cache covert channel," in *DIMVA*, 2015.

[8] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-speed covert channel attacks in the cloud," in *USENIX Security*, 2012.

[9] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *CCSW*, 2011.

[10] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *S&P*, 2015.

[11] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *HPCA*, 2015.

[12] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud," in *USENIX Security*, 2012.

[13] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007.

[14] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *ISCA*, 2012.

[15] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, 1992.

[16] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[17] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: secure dynamic cache partitioning for efficient timing channel protection," in *DAC*, 2016.

[18] F. Liu and R. B. Lee, "Random fill cache architecture," in *MICRO*, 2014.

[19] J. Chen and G. Venkataramani, "CC-Hunter: Uncovering covert timing channels on shared processor hardware," in *MICRO*, 2014.

[20] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *SAC*, 2006.

[21] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *CT-RSA*, 2006.

[22] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *RAID*, 2015.

[23] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel last-level cache," Cryptology ePrint Archive, Tech. Rep., 2015.

[24] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in Intel processors," in *DSD*, 2015.

[25] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," Cryptology ePrint Archive, Tech. Rep., 2015.

[26] M. Payer, "HexPADS: a platform to detect stealth attacks," in *ESSoS*, 2016.

[27] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas, "QuickRec: Prototyping an Intel architecture extension for record and replay of multithreaded programs," in *ISCA*, 2013.

[28] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: a software-hardware interface for practical deterministic multiprocessor replay," in *ASPLOS*, 2009.

[29] A. Burtsev, "Xen deterministic time-travel (XenTT)."

[30] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr, "Abstractions for practical virtual machine replay," in *VEE*, 2016.

[31] G. Altekar and I. Stoica, "ODR: output-deterministic replay for multicore debugging," in *SOSP*, 2009.

[32] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: probabilistic replay with execution sketching on multiprocessors," in *SOSP*, 2009.

[33] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *OSDI*, 2002.

[34] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *SOSP*, 2005.

[35] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *ASPLOS*, 2008.

[36] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: versatile protection for smartphones," in *ACSAC*, 2010.

[37] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Computers & Security*, 2013.

[38] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou, "Detecting covert timing channels with time-deterministic replay," in *OSDI*, 2014.

[39] P. M. Broersen, *Automatic autocorrelation and spectral analysis*. Springer Science & Business Media, 2006.

[40] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira, "Cyrus: unintrusive application-level record-replay for replay parallelism," in *ASPLOS*, 2013.

[41] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *ISCA*, 2008.

[42] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: a full system simulator for multicore x86 CPUs," in *DAC*, 2011.

[43] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "OLTP-Bench: An extensible testbed for benchmarking relational databases," *PVLDB*, 2013.

[44] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[45] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *TCCA*, 1995.

[46] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *CCSW*, 2009.