Understanding and Mitigating Covert Channels Through Branch Predictors

DMITRY EVTYUSHKIN and DMITRY PONOMAREV, State University of New York at Binghamton
NAEL ABU-GHAZALEH, University of California, Riverside

Covert channels through shared processor resources provide secret communication between two malicious processes: the trojan and the spy. In this article, we classify, analyze, and compare covert channels through dynamic branch prediction units in modern processors. Through experiments on a real hardware platform, we compare contention-based channel and the channel that is based on exploiting the branch predictor's residual state. We analyze these channels in SMT and single-threaded environments under both clean and noisy conditions. Our results show that the residual state-based channel provides a cleaner signal and is effective even in noisy execution environments with another application sharing the same physical core with the trojan and the spy. We also estimate the capacity of the branch predictor covert channels and describe a software-only mitigation technique that is based on randomizing the state of the predictor tables on context switches. We show that this protection eliminates all covert channels through the branch prediction unit with minimal impact on performance.

CCS Concepts: • **Security and privacy** → **Security in hardware**; *Systems security*; • **Computer systems organization** → *Architectures;*

Additional Key Words and Phrases: Security, covert channel, branch predictor

ACM Reference Format:
Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and mitigating covert channels through branch predictors. ACM Trans. Archit. Code Optim. 13, 1, Article 10 (March 2016), 23 pages.
DOI: http://dx.doi.org/10.1145/2870636

1. INTRODUCTION

Modern computer systems are typically shared by multiple applications that belong to different security domains. To provide security, systems often have to restrict resources that can be accessible by a program [Yee et al. 2009]. For example, the Android mobile operating system (OS) requires users to explicitly grant permissions for each application. Some classes of applications can be granted access to the network, whereas others can be restricted from it. However, the applications that are restricted from the network access can still be allowed to access sensitive user data.

To illustrate the preceding scenario, consider two applications running concurrently on the same system: a password manager and a weather widget. The password manager should not be allowed to communicate to any application inside or outside of the system

This work is supported by the National Science Foundation under grant CNS-1422401. The statements made herein are solely the responsibility of the authors.
Authors' addresses: D. Evtyushkin and D. Ponomarev, Computer Science Department, 4400 Vestal Parkway East, Binghamton, NY 13902; emails: {devtyushkin, dima}@cs.binghamton.edu; N. Abu-Ghazaleh, Computer Science and Engineering Department, 900 University Avenue, Riverside, CA 92521; email: nael@cs.ucr.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1544-3566/2016/03-ART10 $15.00
DOI: http://dx.doi.org/10.1145/2870636

to avoid password leakage. Although the password manager code can be buggy, or can even contain embedded backdoors, the user passwords will remain secret provided that the OS correctly enforces communication permissions. At the same time, it is essential for the weather widget to have network access enabled to properly support its functionality. One possible threat in this setup is that an adversary controlling both the password manager and the weather widget can use the networking capabilities of the weather widget to send some sensitive information from the password manager to the outside entity, assuming that the password manager and the weather widget can somehow secretly communicate.

This threat model motivates the following question. How can a malicious or a compromised application transfer data to another malicious application in the absence of a direct communication between them? One way to achieve this is to use shared processor resources to create a covert communication channel. We call the two processes that communicate this way a trojan process (the password manager in the example earlier) and a spy process (the weather widget). To transmit sensitive information, the trojan alters the state of a shared hardware resource to intentionally modulate events on that resource in a way recognizable by the spy. On the receiving side, the spy performs measurements to determine how the trojan is accessing the resource, allowing it to receive and decode the modulated events. We present our threat model and assumptions in Section 3.

In this article, we classify, analyze, and comprehensively compare covert channels through processor branch predictor unit. This covert channel is possible because the branch predictor is shared by multiple applications running on the same CPU. Furthermore, the contents of the branch predictor tables are not flushed on context switches. Therefore, when the trojan process modifies the state of the predictor, it impacts the branch prediction rate and the execution time of the spy process (if the spy executes immediately after the trojan, or simultaneously with the trojan). By measuring its own execution time or the branch misprediction rate, the spy can deduce whether the trojan is transmitting a “one” or a “zero” through its manipulations with the predictor logic.

Two different mechanisms for creating a covert channel through branch predictor have been described in the recent literature. The work of Hunger et al. [2015] outlined a *contention-based* covert channel (CC), which (as the name implies) exploits contention between multiple applications over predictor resources. Specifically, this channel is constructed in the following way. To transmit a one, the trojan process creates contention for the branch predictor by executing a large number of branch instructions such that half of them are taken and the other half are not taken. To transmit a zero, the trojan executes no-op instructions, thus creating no contention. The spy process always executes the same code, consisting of branches that are taken with 50% probability. As a result, when the trojan wants to communicate a one, the contention for the branch predictor table causes the execution time of the spy to be higher. When the trojan wants to communicate a zero, there is no contention and the execution time of the spy is lower. This contention-based channel was only described at a high-level in Hunger et al. [2015].

In our preliminary work presented in Evtyushkin et al. [2015], we proposed an alternative covert channel mechanism that is based on exploiting the *residual state* in the branch predictor and not just the contention for its resources. Specifically, to transmit a one, the trojan executes a large number of taken branches, and to transmit a zero, it executes a large number of nontaken branches. The spy always executes a series of taken branches (as in contention-based channel), but a smaller number than the trojan. As a result, when a one is transmitted, the trojan aligns the state of the branch predictor counters with the characteristics of the spy process, causing the spy to have very few mispredictions. On the other hand, when a zero is transmitted, the

predictor counters are put in a state that causes the largest number of mispredictions by the spy. By limiting the number of branches in the spy process so that their predictions are based on the residual state from the trojan (and not the state created by the spy's own execution), a cleaner separation between the transmitted signals of one and zero can be created through this channel. Intuitively, this channel is also more resilient to the external noise, as it does not fundamentally rely on the presence of contention.

Branch predictor covert channels have a fairly large capacity to be a real threat. For example, the recent study of Hunger et al. [2015] estimated that the bandwidth of the branch predictor channel is comparable to other high-speed covert channels, such as those created through caches or the AES hardware. In terms of the absolute numbers, with some optimizations we can achieve the channel capacity of about 100kbps. Clearly, this threat should be considered seriously in the design of future secure systems. To this end, we also propose a software-only mitigation technique that randomizes the state of the branch predictor tables on context switches.

Specific contributions of this article are the following:

—We describe a complete implementation of both CC (introduced in Hunger et al. [2015]) and covert channel based on exploiting residual predictor state (first introduced in Evtyushkin et al. [2015]. We compare both types of channels on the same system in an environment without noise in both the single-threaded and SMT settings.
—We extend this study to account for the noisy environments, where a noise process executes alongside the trojan and the spy processes, and shares the branch predictor with them. Again, we compare the two covert channels side by side and consider several execution schedules that differ in how the spy, the trojan, and the noise process share the execution resources. Our results show that although both channels are effective in clean execution environments (although a residual state channel provides higher signal amplitude), the residual state channel is also realizable in noisy environments, with other unrelated applications running in the background.
—We analyze the capacity of the residual state–based covert channel (RSC) when fast process scheduling between the trojan and the spy is used. Furthermore, we quantify the resulting transmission bitrates and error rates under different channel settings.
—We propose a software-only mitigation mechanism that randomizes the branch predictor state on context switches. We implemented this mechanism inside the Linux kernel and analyzed the sensitivity of performance and security to the number and type of branches that need to be executed on context switches to cause the randomization.

2. DYNAMIC BRANCH PREDICTORS

The branch prediction unit plays a critical role in achieving high performance of today's CPUs, because every branch misprediction results in significant loss of instruction execution opportunities and incurs overhead to undo the side effects of erroneous speculations. This is especially true for deeply pipelined processors with a high degree of superscalarity.

Covert channels described in this article work with any dynamic branch predictor, because the mechanisms for creating covert communication do not require knowledge of the specific predictor details. Although reverse engineering specific predictor configuration can lead to a higher-capacity channel (as the spy and the trojan can precisely target and use specific parts of the prediction table), such advanced explorations are left for future work. For simplicity, we use the *gshare* predictor [McFarling 1993] illustrated in Figure 1 to explain how the branch predictor channels are created. Note

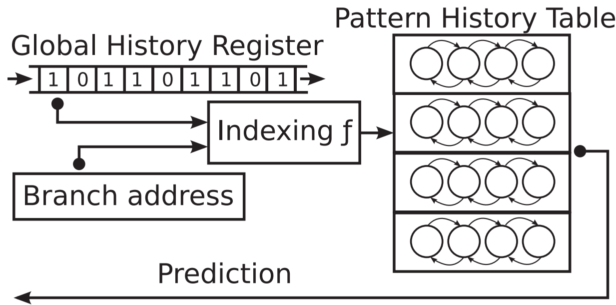


Fig. 1. Schematic of a *gshare* predictor.

that all of our experiments were performed on a real machine equipped with a Haswell processor with specific implementation details of the branch predictor unknown to us.

In a *gshare* predictor, as shown in Figure 1, the Global History Register (GHR) is a shift register that accumulates the history of several recently executed branches. The Pattern History Table (PHT) is a relatively large table of two-bit saturating counters, with the countervalues indicating a prediction range from *strongly not taken* to *strongly taken*. The indexing function XORs the program counter of the branch that is being predicted with the bits from the GHR. Thus, the resulting indexed PHT entry is chosen based on both global and local branch information.

3. THREAT MODEL AND ASSUMPTIONS

We assume that two compromised (or malicious) applications are running in the system: a trojan and a spy. We assume that the trojan is a more privileged program that has access to sensitive data that it attempts to transmit to the spy program. No other communication channels (through the network, shared memory, file system, etc.) exist between the trojan and the spy, and therefore the covert channel represents the only means for these programs to communicate with each other.

We assume that the trojan and the spy are co-located on the same core, either on different SMT contexts or time sharing the use of the core. This assumption is needed because the branch prediction unit is shared on the same physical core but not across different cores of a multicore processor.

The system software is assumed to be uncompromised so that it properly enforces the access control and preserves legitimate information flows. The two processes only require normal user-level privileges. The channel does not require access to performance counters and therefore would work even if these are disabled, as is commonly done on cloud systems [Zhang et al. 2011]. However, if the access to performance counters is available, then a significantly better signal quality can be achieved. In our evaluations, we consider covert channels through both performance counters and execution time.

4. COVERT CHANNEL CLASSIFICATION

In this section, we describe two mechanisms for constructing covert channels through branch predictors, and we demonstrate the code that needs to be executed by the trojan and the spy processes to realize these channels.

4.1. Contention-Based Covert Channels

The first way to create a covert channel through branch prediction unit is to use contention for its resources between the trojan and the spy. To be consistent with prior works [Hunger et al. 2015], we call this type of channel *contention based* and refer to it

```
/* Trojan: */
while(1){
 if (time(0)%2){
   branches();
 }
 else{
   nops();
 }
}

/* Spy: */
for (int i=0; i <
MAX_PROBES; i++){
  usleep(SLEEP_T);
  start_t=rdtsc();
  branches();
  end_t=rdtsc();
}
```
Code for Trojan and Spy

```
branches:
push    %rbp
movl    $0x1,-0x8(%rbp)
cmpl    $0x0,-0x8(%rbp)
jne .L2 # taken #
nop
cmpl    $0x0,-0x8(%rbp)
je .L1
nop
nop
.L2:
cmpl    $0x0,-0x8(%rbp)
je .L1 # not-taken #
nop
nop
......................
.L1:
pop     %rbp
retq
```
Code for branches()

```
nops:
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
......................
```
Code for nops()

Fig. 2. Code used to construct CC.

as CC in this article. The idea and a high-level overview of CC was presented in Hunger et al. [2015].

CC is constructed in the following way. To transmit the value of one, the trojan process executes a large number of branch instructions such that half of them are taken and the other half are not taken. This activity by the trojan creates a random contention for the use of the branch predictor. To transmit a zero, the trojan executes no-op instructions (busy waits), thus creating no contention for the branch predictor. Simultaneously, the spy process always executes the same code, consisting of branches that are taken with 50% probability, again creating contention for the predictor. As a result, when the trojan wants to communicate a one, the contention for the branch predictor increases the number of branch mispredictions and the execution time of the spy. When the trojan wants to communicate a zero, there is no contention for the predictor, and thus the number of mispredictions experienced by the spy and its execution time decrease.

For demonstration, we assume that the trojan sends alternating ones and zeroes. The code for the spy and the trojan processes to implement CC is shown in Figure 2.

4.2. Residual State–Based Covert Channel

Apart from creating branch predictor contention, a covert channel can also be built using the observation that the prediction accuracy of a spy process can be directly impacted (at least for a short period of time) by the residual state of the predictor counters left by the trojan that executed immediately before the spy. We refer to this channel as RSC. If the time duration when the spy measures its branch behavior and/or performance is carefully controlled to magnify the impact of the residual state, a covert channel with an even stronger signal than CC can be created.

In RSC, the contention for the branch predictor unit does not change. To transmit a one, the trojan executes a large number of taken branches, and to transmit a zero, it executes a large number of nontaken branches. The spy always executes a series of taken branches (as in contention-based channel), although it is a smaller number than the trojan. In this case, the predictions for the spy's branches are impacted by the residual state left by the trojan and not by the spy's own prediction history buildup. As

a result, when a one is transmitted, the trojan aligns the state of the branch predictor counters with the characteristics of the spy process, causing the spy to have very few mispredictions. On the other hand, when a zero is transmitted, the predictor counters are put in a state that causes the largest number of initial mispredictions by the spy.

An important aspect of RSC is that the spy's code is not executed constantly. Instead, it is only executed once for each probing period, recording the timestamp counter, or reading the branch misprediction performance counter. When the spy completes the execution of its block of branches, it executes the `sleep()` function for a predetermined amount of time to allow the trojan to refill the predictor table. After sufficient time is given to the trojan to refill the predictor state, the spy executes its block of branches again. The spy samples the execution time or the performance counter readings five times a second in the presented experiments, with the trojan changing the transmission from a one to a zero every second. The duration of the block of branches executed by the spy on every sample is carefully chosen to ensure that the branch predictions performed within that block are affected by the state created by the trojan and not by the spy's own history. In the presented experiments, we set this number to 500,000 branches, because we observed the best channel characteristics with this setting.

Transmitting data through the branch predictor state in this manner in a single-threaded environment is possible because the PHT contents are not flushed on a context switch. Several branch predictor designs [Evers et al. 1996] have been introduced that considered context switches that erase the branch history data from the old context in the PHT. However, these designs have not been adopted in commercial products, because no performance benefits were observed [Co and Skadron 2001].

Both CC and RSC can also be created on a simultaneously multithreaded (SMT) processor core. The SMT cores share the same branch predictor hardware and its data structures among the threads. Although it is possible to design a branch predictor with split data structures for the simultaneous threads, such splitting does not bring significant performance improvements [Ramsay et al. 2003] and thus is not typically used. We demonstrate and compare CC and RSC in both single-threaded and multithreaded environments.

We also observed that adding a uniformly distributed number of no-op instructions between consecutive branches improves the amplitude of the covert channel measured by the spy, as it increases the number of affected PHT entries. If the branch predictor hashing function can be reverse engineered, the PHT priming can be done even more effectively. The code for the trojan and the spy process to implement RSC is shown in Figure 3.

5. ANALYZING CC AND RSC IN A CLEAN ENVIRONMENT

We demonstrate and evaluate covert channels presented in this article on a real hardware platform. All of our experiments were performed on a machine with an Intel Core i7-4800MQ CPU (Haswell microarchitecture) clocked at 2.7GHz. The machine has 16GB of DDR3 memory clocked at 1,600MHz. We consider scenarios with and without SMT—to evaluate the latter, we disabled the SMT support. The machine runs a Ubuntu 14.04.2 LTS OS, with a generic GNU/Linux kernel version 3.16.0-31.

This section presents the results in a clean environment, where we ensure that only the trojan and the spy execute on the core. Moreover, we tightly control the scheduling of these two processes to create ideal conditions for a covert channel. In the next section, we relax these conditions and compare both types of covert channels in a noisy environment.

As a measurement mechanism, the spy can use the branch-related performance counters or its own execution time. Depending on the measurement used, the channel can be classified as either a storage channel or a timing channel [Gligor 1993; Wray 1991].

```
/* Trojan: */
while(1){
 if (time(0)%2){
   taken();
 }
 else{
   nottaken();
 }
}

/* Spy: */
for (int i=0; i <
MAX_PROBES; i++){
  usleep(SLEEP_T);
  start_t=rdtsc();
  taken();
  end_t=rdtsc();
}
```

Code for `Trojan` and `Spy`

```
taken:
push    %rbp
movl    $0x1,-0x8(%rbp)
cmpl    $0x0,-0x8(%rbp)
jne .L2
nop
nop
nop
.L2:
cmpl    $0x0,-0x8(%rbp)
jne .L3
nop
nop
nop
.L3:
cmpl    $0x0,-0x8(%rbp)
jne .L4
nop
......................
```

Code for `taken()`

```
nottaken:
push    %rbp
movl    $0x1,-0x8(%rbp)
cmpl    $0x0,-0x8(%rbp)
je .L1
nop
cmpl    $0x0,-0x8(%rbp)
je .L1
nop
nop
nop
cmpl    $0x0,-0x8(%rbp)
je .L1
nop
nop
......................
.L1:
pop     %rbp
retq
```

Code for `nottaken()`

Fig. 3. Code used to construct RSC.

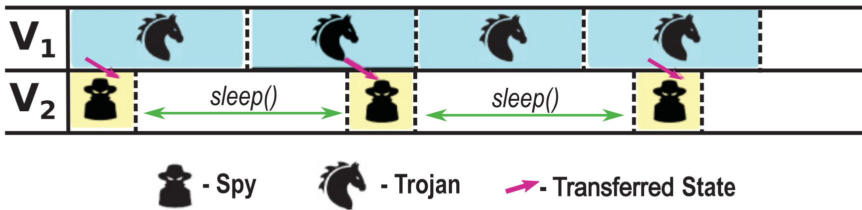


Fig. 4. Scheduling of the trojan and the spy in SMT mode.

Although using performance counters provides higher measurement accuracy, it may require administrative privileges from the spy. Whether such privileges are required or not depends on the particular hardware, OS, and even hardware configuration. For example, according to the Intel's *Architecture Software Developers Manual* [Intel 2010], a particular configuration set allows or disallows user-level accesses to performance counters. However, we conservatively assume that performance counters are not always available and also consider timestamp counters as a measurement mechanism for the spy.

5.1. Covert Channels in SMT Mode

Our first set of experiments includes demonstration of the two branch predictor covert channels in an SMT setting, where the spy and the trojan execute concurrently. For the experiments under the SMT conditions reported here, we assign both processes to isolated virtual cores (a single SMT-enabled physical core is represented in the OS as two virtual cores). In this case, the trojan and the spy execute on the same physical core but on different virtual cores. Such a setting allows the processor to fetch instructions simultaneously from two threads. Figure 4 depicts the scheduling of the spy and the trojan in such a scenario.

Figure 5 compares the results of CC and RSC in an SMT setting in a clean execution environment without noise or interference. The x-axis represents the number of seconds from the moment the spy process starts probing the PHT. The y-axis shows the number

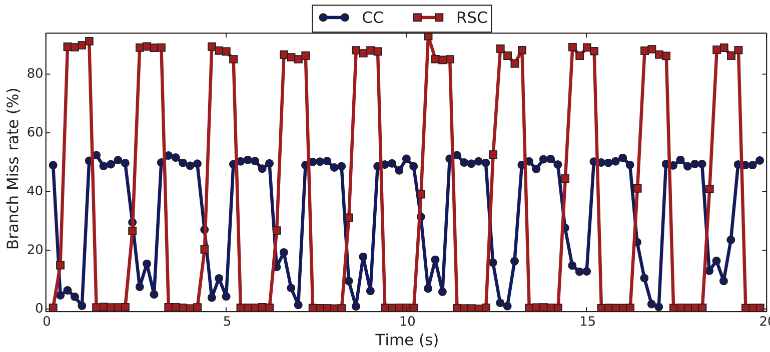


Fig. 5. Comparing branch misprediction–based CC and RSC in an SMT scenario with clean conditions.

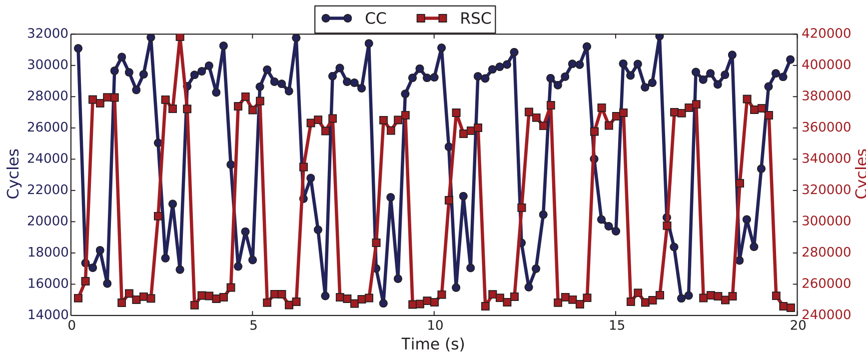


Fig. 6. Comparing execution time–based CC and RSC in an SMT scenario with clean conditions.

of branch mispredictions measured in each sample by the spy in CC and RSC. For this experiment, the trojan continuously executes a large block of branches (we used 500,000 branches in each block) or no-ops, depending on the requirements of each channel, as described previously. Each block of branches executed by the spy process contains 10,000 branches for RSC and 1,000 branches for CC—we found that these values provide the most stable signal for each channel. In each case, the spy samples the prediction accuracy (or its execution time) five times a second. The trojan transmits a zero during even seconds and a one during odd seconds.

As shown by the graph, both CC and RSC are quite effective covert channels under this scenario, with clear separation between the levels of one and zero. However, as expected, the amplitude of the signal is higher with RSC, because RSC is explicitly reusing the leftover state from the trojan instead of relying on contention. Specifically, in CC, the low signal level corresponds to a 6% misprediction rate, and the high signal level corresponds to about a 50% misprediction rate on average. For RSC, the signal levels are 0.4% and 86%, respectively, providing a significantly higher amplitude of the channel signal.

Covert channels shown in Figure 5 can only be created if the spy has access to performance counters, which may not always be possible on all systems. When such access is not possible, the spy has to rely on measuring its own execution time. Figure 6 depicts the waveforms obtained by both CC and RSC if only the execution time can be measured by the spy. All settings of the spy and the trojan are identical to what was described earlier. Note that this figure includes two y-axes. The left y-axis corresponds to CC, and the right y-axis corresponds to RSC. Although the shapes of the two waveforms

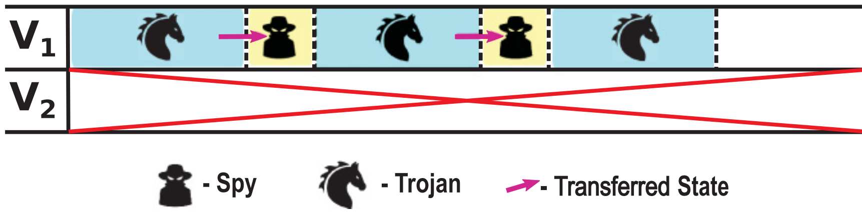


Fig. 7. Scheduling of the trojan and the spy in single-threaded mode.

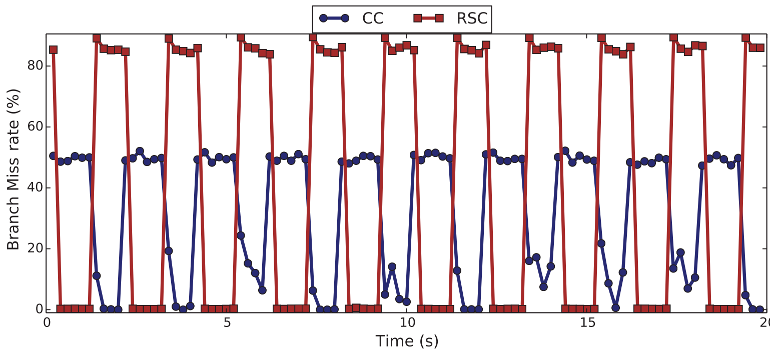


Fig. 8. Comparing branch misprediction–based CC and RSC in a single-thread scenario with clean conditions.

are quite similar (and both channels are effective), there is a larger absolute difference between the levels of one and zero for the RSC channel, potentially making the RSC channel more resilient to external noise.

5.2. Covert Channels in Single-Threaded Mode

Next, we consider the creation of CC and RSC in single-threaded execution mode, where instead of executing simultaneously, the trojan and the spy are taking turns being scheduled on the same CPU core. In this section, we only consider the case when the trojan and the spy are scheduled consecutively and are the only two processes using the core. We defer the treatment of more noisy environments until the next section.

For these experiments, we achieve consecutive scheduling of the trojan and the spy by dedicating a physical CPU core only to these two programs, using the default OS functionality. The ideal scheduling depicted in Figure 7 is achieved in this case. The trojan executes continuously, and the spy only executes periodically and immediately after the measurement relinquishes the rest of its time slice. Specifically, the spy interrupts the trojan's execution, samples the PHT, and switches the execution back to the trojan. The contention in single-threaded CC is different from the SMT-based CC. When the trojan executes a block of branch instructions, it fills the predictor tables with the direction information valid for these branches. As a result, when the spy executes its own branch instructions, it experiences a higher number of mispredictions. When the trojan does not execute branches, no contention is created and the spy reuses its own information accumulated in the predictor. We note that no changes are required in the source code of the trojan or the spy to explicitly adjust them to SMT or single-threaded modes.

Figure 8 shows the results comparing CC and RSC waveforms obtained by measuring the branch misprediction rate of the spy in a single-threaded execution environment. Figure 9 compares CC and RSC measured in terms of the execution time in a

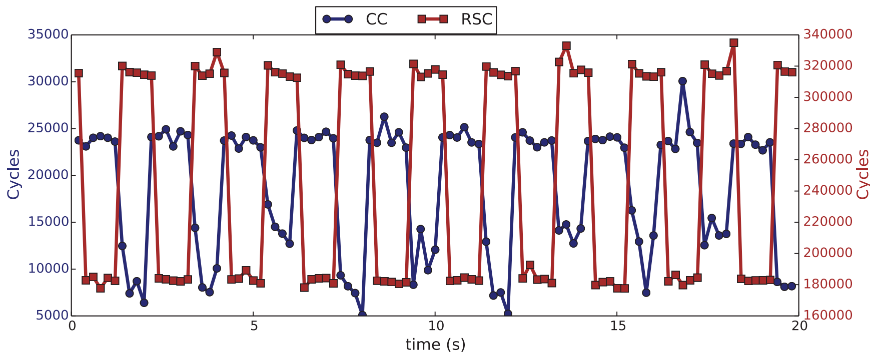


Fig. 9. Comparing execution time-based CC and RSC in a single-threaded scenario with clean conditions.

single-threaded environment. As can be seen from these results, both channels are quite effective in a clean environment without noise processes.

6. ANALYZING CC AND RSC IN A NOISY ENVIRONMENT

The previous section demonstrated that both CC and RSC are effective secret communication channels in clean environments. In this section, we consider the impact of noise and interference from other programs on the robustness of these covert channels. As before, we consider both SMT and single-threaded execution environments. As the source of noise, we consider the GCC compiler compiling a Linux kernel. In this scenario, GCC is an integer benchmark that exhibits complex branch behavior and can significantly distort the state of the prediction table, thus complicating the communication between the trojan and the spy. GCC compiler is a highly CPU-bound noise process—the average CPU utilization during kernel compilation was 91.56%.

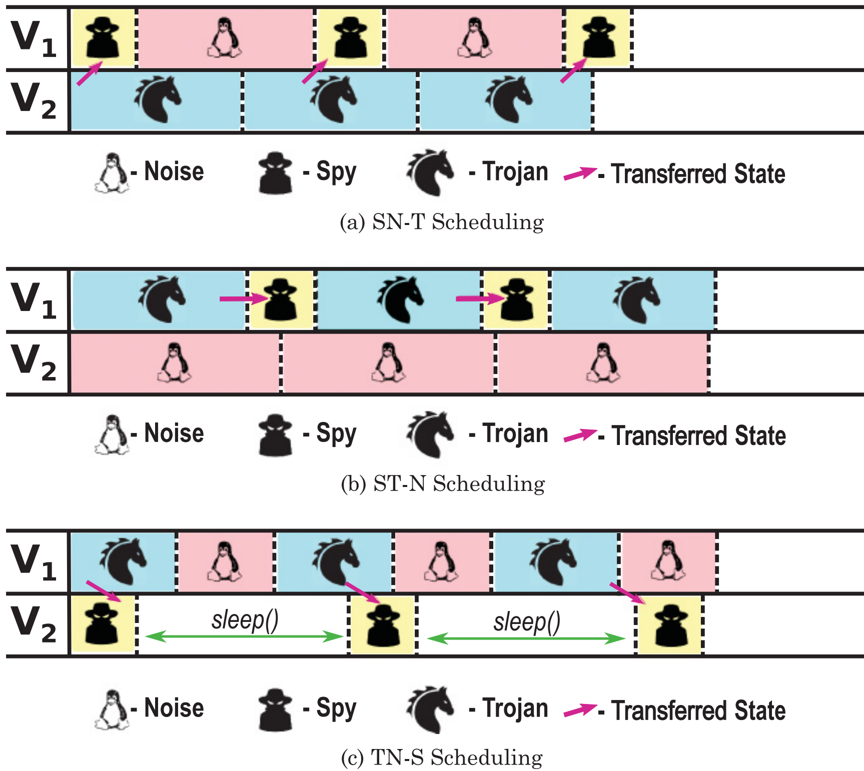
6.1. RSC Under Noise

In this section, we analyze RSC properties in a noisy environment. We consider both SMT and single-threaded settings. First, we consider SMT cores and examine the following three execution schedules, which can be realized by appropriately setting the affinity masks of the trojan, the spy, and the noise processes. We assume two virtual cores (**V1** and **V2**) and one physical core. We refer to the trojan process as **T**, the spy process as **S**, and the noise process as **N**.

—**Schedule SN-T: S** and **N** execute on **V1**, and **T** executes on **V2**. In this case, the trojan has the entire thread context (virtual core) to itself, whereas the noise and the spy alternate execution on the other context.
—**Schedule ST-N: T** and **S** execute on **V1**, and **N** executes on **V2**. Here, the noise process executes all the time, whereas the trojan and the spy alternate.
—**Schedule TN-S: T** and **N** execute on **V1**, whereas **S** executes on **V2**.

These schedules are demonstrated in Figure 10. Furthermore, we also consider RSC in the noisy environment in a single-threaded scenario, **Schedule STN**: **S**, **T** and **N** execute consecutively on the same core.

Figure 11 presents the results of RSC under the three SMT schedules and the single-threaded schedule. The channels are shown both in terms of the number of branch mispredictions and in terms of the execution time. For each scheduling type, we also run the experiment without executing the trojan. In this case, the spy's measurements is only impacted by the background noise in the system. A good channel would have the *noise* and the *signal* levels easily distinguishable. This property not only makes it pos-



(a) SN-T Scheduling

(b) ST-N Scheduling

(c) TN-S Scheduling

Fig. 10. Scheduling of noise, trojan, and spy processes.

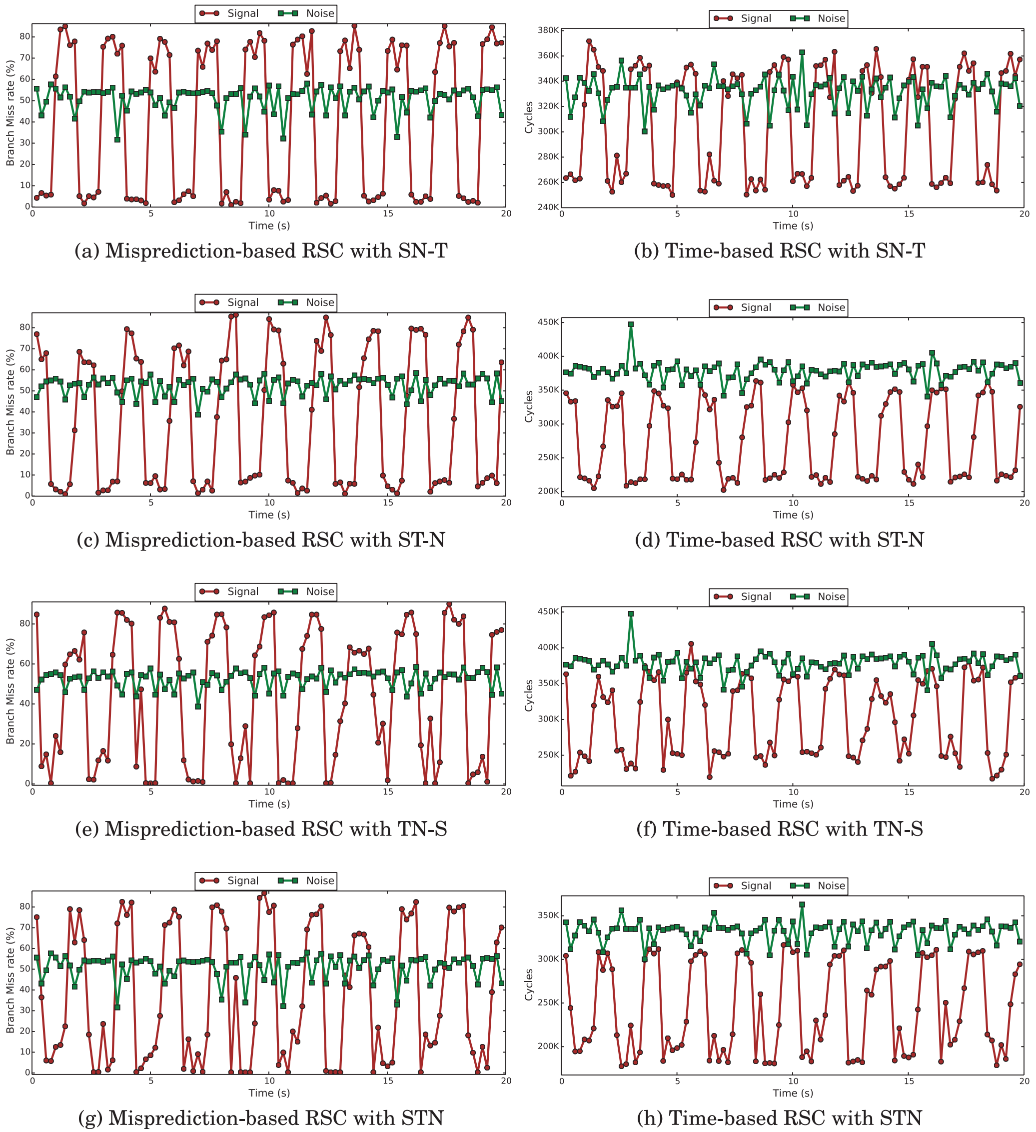
sible to tolerate the system noise but also can contribute to building an asynchronous channel with no prior synchronization between the trojan and the spy. This becomes possible because the spy can explicitly detect when information is being transmitted over the channel.

Figure 11(a) and (b) show the RSC waveforms for branch mispredictions and cycles, respectively, as measured by the spy, for the **SN-T** schedule shown earlier. Figure 11(c) and (d) plot similar results for the **ST-N** schedule. Finally, Figure 11(e) and (f) show the RSC channel for the **TN-S** schedule. As seen from the results, the channels are visible and effective for each execution schedule, both for the number of mispredictions and for the execution cycles. In addition, the channel is clearly distinguishable from the noise pattern and therefore can be created even in the presence of external noise.

Figure 11(g) and (h) show the RSC waveforms obtained by measuring branch mispredictions and execution cycles in a single-threaded schedule (**STN**). As with SMT, for comparison purposes we also show the channel between the background noise process (GCC compiler) and the spy when only two of them are executing. As seen from the results, RSC in a single-threaded scenario is also easily distinguishable from the background noise.

6.2. CC Under Noise

Next, we performed similar experiments with CC. Figure 12 shows these results. Specifically, Figure 12(a) through (f) show results for CC for the three SMT schedules listed earlier. As in the RSC case, the CC channel is compared against a hypothetical channel between the noise process and the spy. In contrast to what was observed for RSC,

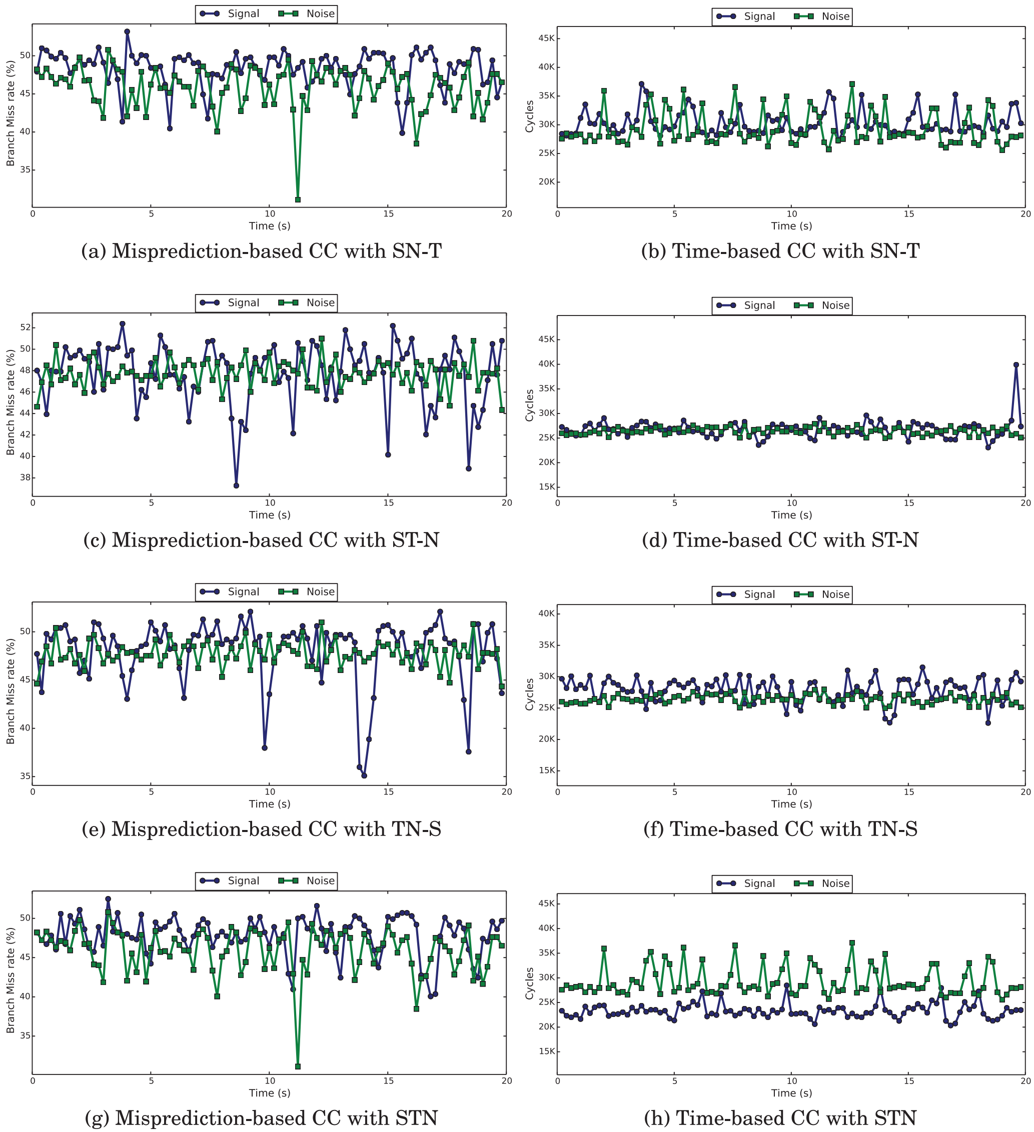


(a) Misprediction-based RSC with SN-T
(b) Time-based RSC with SN-T
(c) Misprediction-based RSC with ST-N
(d) Time-based RSC with ST-N
(e) Misprediction-based RSC with TN-S
(f) Time-based RSC with TN-S
(g) Misprediction-based RSC with STN
(h) Time-based RSC with STN

Fig. 11. RSC waveforms in a noisy environment and comparison with background noise.

the channel created by CC is practically indistinguishable from noise. Therefore, CC cannot be effectively constructed in the presence of noise and interference, regardless of a particular schedule between the spy, the trojan, and the noise process. The main reason is that CC fundamentally relies on the lack of contention to transfer one of the possible values (either a one or a zero). However, the presence of noise process practically eliminates the noncontention execution periods.

Figure 12(g) and (h) show CC waveforms for the single-threaded schedule. Again, as with the SMT scenario, the CC waveform is practically indistinguishable from the background noise (created by the GCC compiler).



(a) Misprediction-based CC with SN-T (b) Time-based CC with SN-T

(c) Misprediction-based CC with ST-N (d) Time-based CC with ST-N

(e) Misprediction-based CC with TN-S (f) Time-based CC with TN-S

(g) Misprediction-based CC with STN (h) Time-based CC with STN

Fig. 12. CC waveforms in a noisy environment and comparison with background noise.

In conclusion, whereas both CC and RSC are effective communication channels in a clean environment where only the trojan and the spy execute, only RSC can provide a reliable channel in the presence of noise.

7. COVERT CHANNEL CAPACITY ESTIMATION

Covert channel practicality is often determined by its capacity. When a covert channel is used to transfer only a small amount of data (e.g., cryptographic keys), its capacity may be secondary to reliability and noise resilience. However, only high-capacity channels are useful for transmitting large amounts of data. For example, it would take 500 days

to transmit an image file of 4MB using a thermal covert channel with a very low capacity [Guri et al. 2015].

The capacity of a covert channel is impacted by implementation-specific details and optimizations. The transmission bitrate depends on the nature of the shared resource used and on a particular data transmission protocol. Some channels can be noisy requiring noise reduction techniques and error correction codes, such as Hamming codes [Hamming 1950].

In this section, we explain our experimental setting and estimate the transmission bitrate and error rate for the channel described in this article.

7.1. Capacity Estimation of RSC

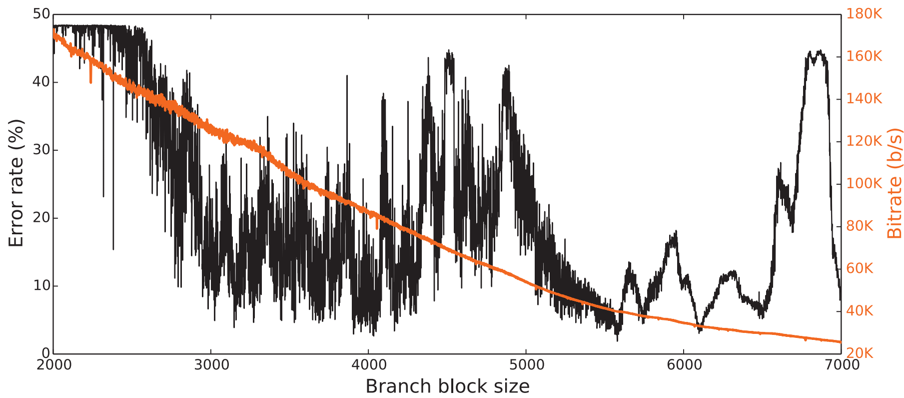
Computing the maximum possible covert channel capacity would require a large number of optimizations and knowledge about the exact implementation of the branch predictor unit. Instead, we construct a simple and fast covert channel prototype that provides a reasonable estimate for the channel capacity. In the slow channel, the trojan changes the signal level once every second, and therefore the channel capacity is only one bit per second. In the fast channel, the execution order switches between the trojan and the spy as fast as possible.

In particular, both programs rely on the `sched_yield()` function, which relinquishes the rest of the CPU time slice allocated to a process. Another important difference is that instead of transmitting alternating ones and zeroes, now the trojan transmits a sequence of randomly generated bits.

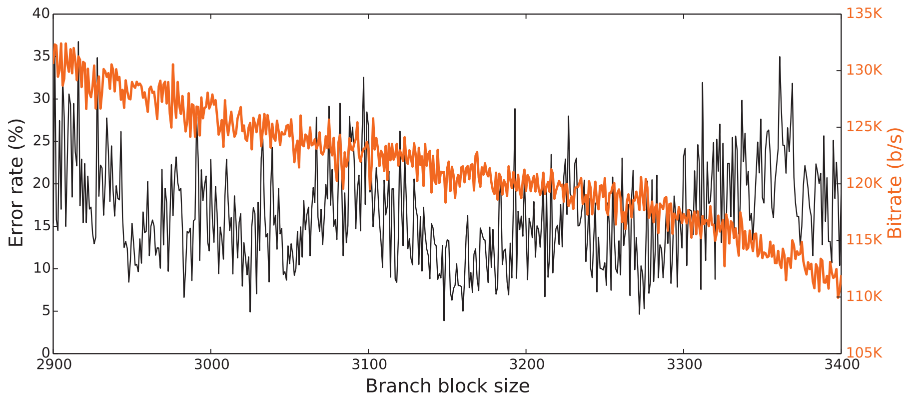
Every time the trojan is scheduled, it executes one of the two code blocks with branch instructions, priming the branch predictor. To determine which code block to run, the trojan looks up the array consisting of randomly generated bits. The bits in this array determine the type of signal the trojan sends during the current communication cycle. After that, the trojan calls the `sched_yield()` function to force a context switch and subsequent scheduling of the spy process. The spy probes the branch predictor by executing a block of taken branches. The spy also measures the branch misprediction rate, or the time that it takes to execute this block of code. Based on these measurements, the spy can determine whether the trojan transmitted a one or a zero. Following that, the spy calls the `sched_yield()` function to force the context switch to the trojan again. Since both processes always have code to run, they create constant demand for the CPU resources. As a result, the OS does not schedule other processes on that core, allowing the branch predictor state to transfer from the trojan to the spy. After the transmission completes, we compare the number of correctly transmitted bits and the number of errors to calculate the error rate.

The size of branch blocks executed by the spy and the trojan can be adjusted to control the channel efficiency. The number of branches should be large enough to affect most of the branch predictor table entries but small enough to prevent excessive usage of entries already affected by the block. In addition, if the trojan executes a very large code block, it gives the OS a sufficient time to generate the timing interrupt and perform a context switch, thus distorting the schedule. To maintain channel-friendly scheduling, we enforce that each process executes the same number of branches. The optimal number of branches is a processor-specific parameter, which depends on the configuration of the prediction unit.

To estimate the channel's capacity and the error rate, we transmitted 1,000 randomly generated bits and measured the error rate in the signal received by the spy. In addition to that, we measured the time spent receiving the signal, and from there we estimated the channel capacity. We ran the experiment for different sizes of branch code blocks used by the trojan and the spy. In particular, we started from code blocks of 2,000 instructions and finished with blocks of 7,000 branches. We computed the channel



(a) Bitrate and error rate for different branch instruction block sizes (from 2,000 to 7,000)



(b) Bitrate and error rate for branch blocks from 2,900 to 3,400 instructions

Fig. 13. Capacity (bitrate) and error rate for channels created with different branch code block sizes.

capacity and the error rate for each configuration. We ran the experiment 100 times for each block size and present the results averaged across these 100 experiments.

The runs results of these experiments are presented in Figure 13(a). Using blocks of size less than 2,900 branches does not result in the creation of a usable channel. A more detailed representation of the most interesting region (between block sizes of 2,900 and 3,400) is shown in Figure 13(b). The channel becomes more stable when the block size approaches 4,000 branches. An example of a sweet spot in terms of the trade-off between the bitrate and error rate is the block size of 3,148 branches. In this case, we achieve the average bitrate of 121kbps and an average error rate of 3.9%. In general, we observed oscillations of the error rate as the block size increases. This is a function of specific branch predictor access patterns by these codes (which we did not reverse engineer) and varying amount of system noise.

Presented results show that a fast covert channel can be constructed using branch predictor tables. The resulting channel has desirable properties, as it is fast enough to transfer large amounts of data and has acceptable error rates. Our experiments also demonstrate that a deep knowledge about the branch predictor organization is not required to construct a fast and reliable covert channel. Instead of reverse engineering the branch predictor, a reasonable approach is to experiment with different sizes of the branch code blocks, as presented in this section.

7.2. Improving the RSC Capacity

The capacity of RSC can be further improved. If an adversary is equipped with the knowledge about the branch predictor organization, this can significantly improve the bitrate and reduce the error rate. In this case, instead of manipulating large blocks of branch code to increase the probability of putting the prediction table in the desired state, the trojan can directly target specific table entries. However, such a protocol will be limited to a particular CPU, whereas the statistical channel is not. Although it is outside the scope of this article to analyze such optimizations, we outline several possibilities for the trojan and the spy to improve the channel:

(1) The branch predictor indexing function and the size of prediction structures can be reverse engineered. The adversary can manipulate the branch addresses and the GHR state to force the mappings of branches to desired PHT entries.
(2) The adversary can control the OS scheduler to obtain a CPU quantum of desired length and schedule processes in desired order on any core.
(3) The adversary can access measurement tools, such as the timestamp counter or performance counters, with minimal latency.
(4) The adversary can achieve perfect synchronization between the trojan and the spy at the granularity of a single instruction.

The detailed exploration of these optimizations is left for future work.

8. MITIGATING BRANCH PREDICTOR COVERT CHANNELS

In this section, we describe and evaluate a protection technique that mitigates covert channels through the branch predictor, including CC and RSC.

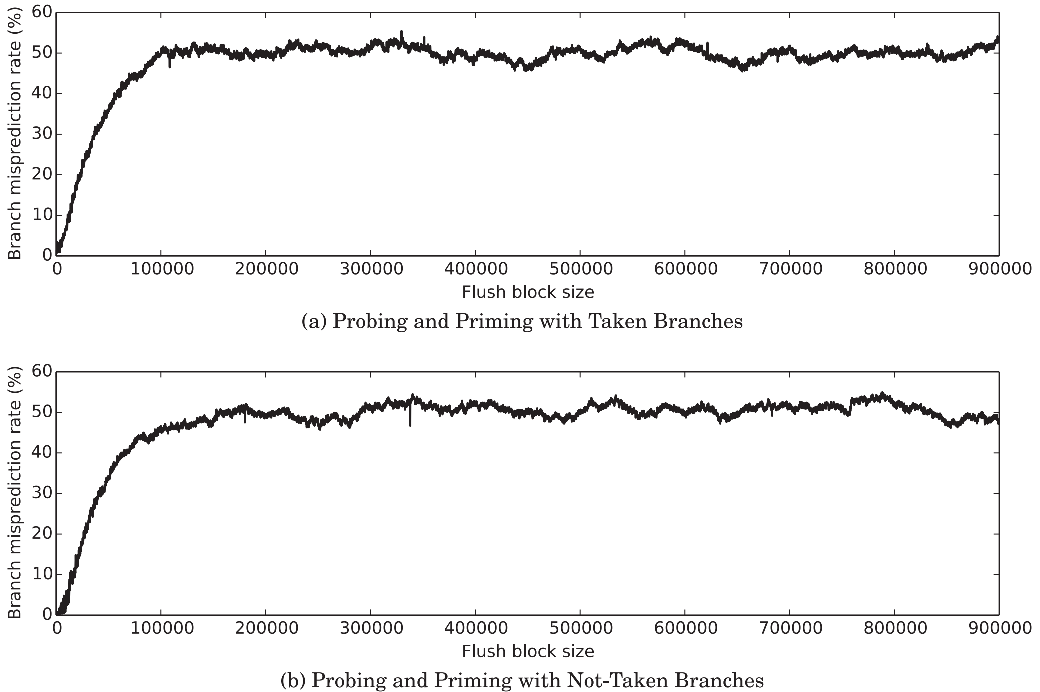
8.1. Channel Mitigation: Flushing the Predictor on Context Switches

To close the channel, we propose a software-only solution, which flushes the branch predictor (or randomizes its state) on context switches. This approach mitigates both types of covert channels considered in this article but by different means. RSC is eliminated because flushing of the branch predictor makes it impossible to place the predictor into one of the desired states. CC is mitigated because the context switch creates constant pressure on the predictor, thus making it impossible to alternate high and low contention stages.

To implement this protection, we modified the context switch routine in the OS kernel. In particular, before the scheduler assigns the next ready process to the CPU, a large block of branch-intensive code is executed to randomize the branch predictor state. As a result, the newly scheduled process starts execution with a clean predictor state. This mechanism effectively eliminates the secret data transmission between the trojan and the spy. Note that this mitigation technique does not consider the spy and the trojan running on two hardware thread contexts of an SMT processor. For security reasons, the OS should not schedule processes from different security domains simultaneously on the same physical core. Alternatively, the SMT support can be disabled.

8.2. Optimal Number of Branches in Randomization Code

The branch predictor can be flushed on a context switch in several ways. For example, a large number of taken or not-taken branches can be executed to put the entire table of prediction counters in one of the strong states. Alternatively, a large number of branches with a random taken/not-taken pattern can be executed, resetting the branch predictor to a neutral state with 50% taken probability. Such randomized approach is more likely to have lower performance impact due to the bimodal nature of branch outcomes.



(a) Probing and Priming with Taken Branches

(b) Probing and Priming with Not-Taken Branches

Fig. 14. Branch predictor flushing using different numbers of random branch instructions.

To determine the optimal number of branch instructions required to randomize the branch predictor in a secure manner, we conducted the following experiment. First, we executed the code, consisting of 1 million branch instructions, several times, thus placing the branch predictor into one of the strong states. We call this phase the *priming* of the predictor. Then, the *flushing* code block was executed once. We varied the number of instructions in the flushing block. Finally, we executed 1,000 branch instructions with the same outcome as branches in the priming phase and measured the number of mispredictions in this block. We call this phase *probing*. The number of branch mispredictions encountered in executing the probing code indicates how well the flushing phase resets the predictor.

First, we considered the flushing code composed of randomized blocks of branch instructions. The results are presented in Figure 14. Figure 14(a) shows how effectively the predictor state is reset after being primed with all taken branches, and Figure 14(b) depicts similar results for priming with not-taken branches. As expected, the small blocks of flushing code are not sufficient to reset the predictor, and the misprediction rate is very low. As the flush code block increases in size, the misprediction rate of the probing phase also increases. The growth stops at about 50%, indicating that the branch predictor tables are reset.

To ensure higher probability of a complete branch predictor reset, the OS needs to use larger flush code blocks compared to the minimal block size that provides 50% misprediction rate of the probing code. To protect against more sophisticated and intelligent adversaries (which could potentially explore even minuscule deviations in branch misprediction rates to detect transmission patterns), we conservatively selected 300,000 branch instructions in the flush block for further experiments and analysis. Therefore, all performance overheads are presented under this very conservative assumption.

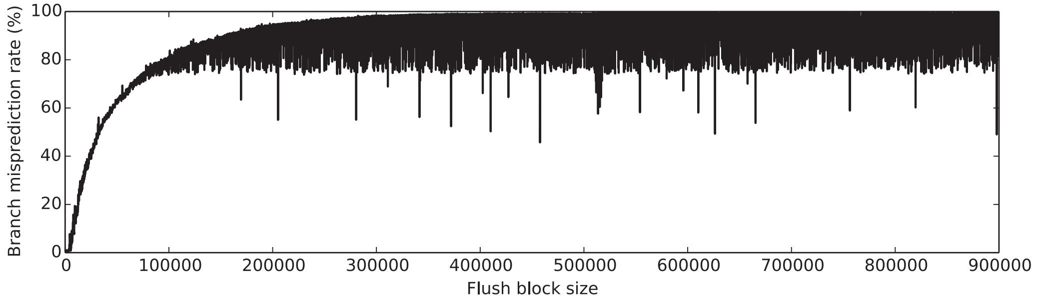


Fig. 15. Branch predictor flushing using taken branches.

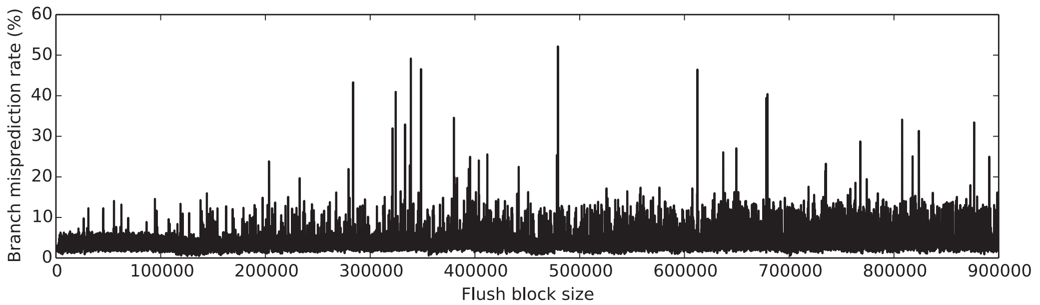


Fig. 16. Branch predictor flushing using not-taken branches.

Next, we examined the effectiveness of using a large number of taken branches as flushing code block. In particular, priming and probing code contains only not-taken branches, and the flushing code contains only taken branches. The results are presented in Figure 15. As expected, larger flush blocks make stronger impact on the branch predictor. However, we note that it is difficult to place all predictor entries into the strongly-taken state, thus causing the misprediction of all probing branches. Furthermore, there are elements of randomness that are present in this flushing mechanism, so the same number of executed instructions can affect the prediction rate differently. This is manifested by the dispersion of the results as the misprediction rates get closer to 100%.

Finally, Figure 16 shows the results of flushing the predictor using not-taken branches. Surprisingly, not-taken branches have a much smaller impact on the branch predictor state. In particular, when the flushing code runs for the first time, it affects the number of mispredictions in priming code significantly. However, the predictor makes adjustments, and the misprediction rate soon decreases. The misprediction rate remains relatively low (below 15%) even for very large blocks of all not-taken branches.

The results presented earlier show that executing an even mix of taken and not-taken branches with a random pattern is the most stable way to reset the predictor. In addition, assuming bimodal behavior of branches under a normal execution pattern, placing the entire branch predictor into one of the strong states results in a higher number of mispredictions. In the rest of the experiments, we assume a random mix of taken and not-taken branches in the flushing code.

Since the protection goal is to place the branch predictor into a neutral state, it is important to avoid flushing the predictor with code that has fixed branch pattern. Otherwise, the predictor will accumulate statistics for those branches and the flushing code will create constant branch pressure instead of randomization of the predictor's

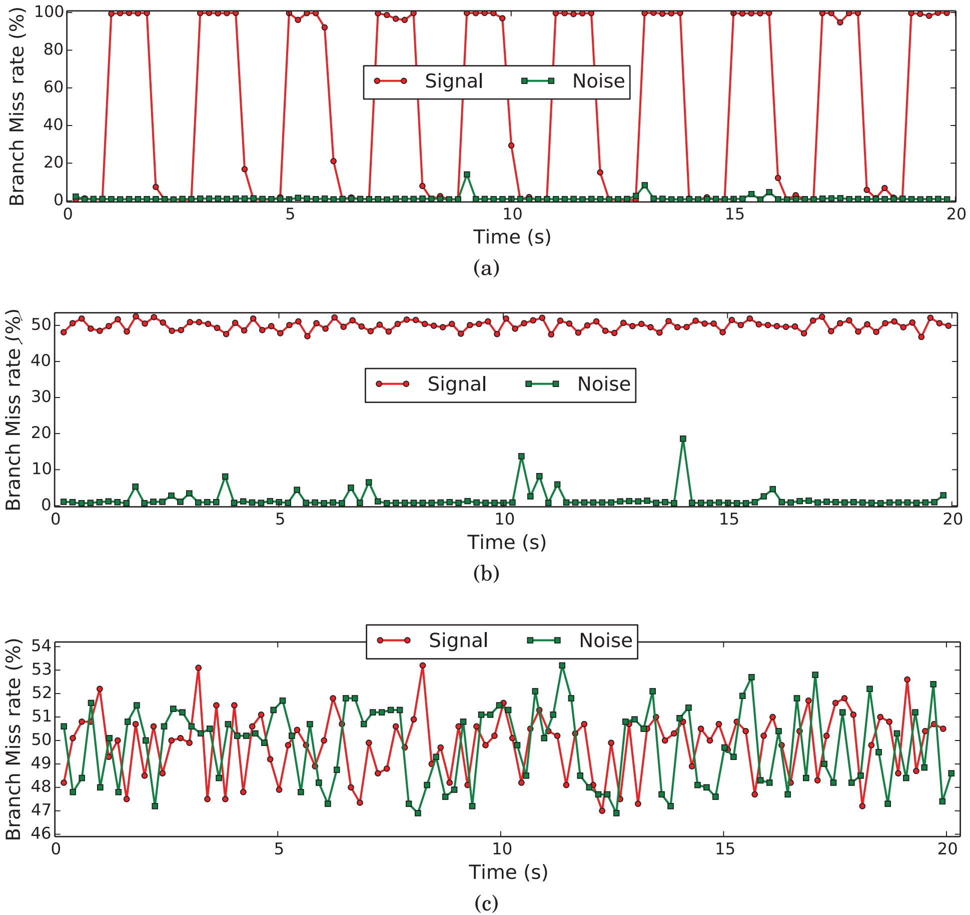


Fig. 17. Channel waveform without protection (a); channel waveform when protection is applied after the trojan is switched out (b); channel waveform when protection is applied after the spy is switched in (c).

state. However, randomizing the pattern of flushing branch instructions on every context switch will result in a significant performance loss. To optimize this process, we generated several randomized blocks of flush code and randomly pick one block to use on every flush instance. This approach provides randomness while minimizing performance impact.

8.3. Results and Performance Overhead

The protection mechanism described earlier has been implemented inside the Linux kernel. Since the covert channel involves the trojan and the spy, there are two options of how the protection can be enforced. The first option targets the trojan (the transmitter of the data). In this case, the protection can be invoked each time the trojan context is switched out. The second option targets the spy (the receiver of the data). Here, the protection can be invoked every time the spy's context is switched in. We implemented both schemes, and the results are presented in Figure 17.

The signal line shows the misprediction rate measured by the spy when the trojan is active, and the noise line represents normal ambient noise measured by the spy with no trojan present. Both protection schemes make covert communication through

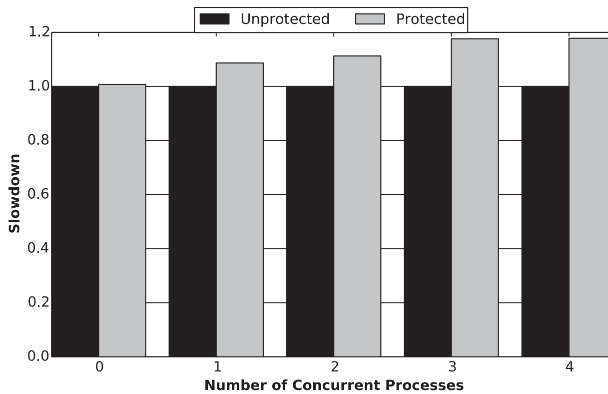


Fig. 18. Performance impact from flushing the branch predictor on context switches.

branch predictor impossible, as the transmission signal waveform can no longer be visible by the spy. The first protection scheme (targeting the trojan) has lower performance overhead. Typically, the number of applications that manage secret data in a system (possible trojans) is much less than the number of applications that have communication permissions (possible spies).

Flushing the branch histories on context switches can have little performance impact, or it may even be beneficial in some cases [Evers et al. 1996; Co and Skadron 2001]. However, the flushing operation itself is expensive when it is performed in software. To completely remove the residual state from the predictor, large blocks of branch code need to be executed. For the experiments described earlier, we used 300,000 branch instructions. We note that since the branch prediction implementation details are unknown, the protection mechanism does not guarantee flushing of all branch predictor entries. However, it makes it extremely hard, if not impossible, for the attacker to probabilistically manipulate the predictor to pass information using RSC or CC. On our experimental system, the overhead of flushing the predictor in software corresponds to the additional latency of 1.2 milliseconds added to the context switch. In addition, the flushing code pollutes the state of caches and branch predictors.

Several techniques can be used to reduce the performance impact. First, instead of flushing the predictor on every context switch, the OS can do so only when there is a threat of undesired information transfer through the predictor. Another optimization is for the OS to group processes by security domains. Predictor flushes are only needed when a context switch happens between processes residing in different security domains. By changing the scheduler algorithm, the OS can minimize the number of context switches that require flushing to decrease performance overhead. Such technique is known as lattice scheduling [Hu 1992].

To evaluate the performance impact of our mitigation technique, we used the CPU performance benchmark from the Sysbench [Kopytov 2004] benchmark suite. We executed the benchmark with protection disabled (no branch predictor flushes) and enabled (branch predictor is flushed before the benchmark's context is switched in). To evaluate the performance impact under various concurrency scenarios, we executed the benchmark alongside several background processes. The background processes were created by executing the same benchmark with disabled protection, and the execution was performed on the same core. This created a higher level of contention for CPU resources, thus increasing the number of context switches.

Figure 18 presents normalized results showing the performance impact of enabled protection. When no other processes are running on the same core, the performance

impact is very small. As more noise processes add contention for the CPU, the OS is forced to perform context switches more often. The performance impact increases, but it does not exceed 20% in our test case. Such overhead is significantly smaller compared to some of the earlier presented solutions for other types of timing channels, such as the shared memory controller channel [Wang et al. 2014b]. We consider software protection as a temporary countermeasure until developers implement branch predictor flushing mechanism in hardware.

9. RELATED WORK

Covert channels through shared microprocessor resources have been explored in several recent efforts. Wang and Lee [2006] presented covert channels using exceptions on speculative load instructions and shared functional units on SMT processors. Wu and Wang [2012] described a covert channel that is based on the Intel Quick Path Interconnect (QPI) lock mechanism. Ristenpart et al. [2009] presented a cross-VM covert channel that exploits the shared cache. Covert channels based on the use of memory bus were presented in Saltaformaggio et al. [2013]. Wang et al. [2014a] presented a covert channel through shared memory controllers and proposed some techniques to close it. Their solution to eliminate interference across security domains is based on per-domain queuing structure and static allocation of time slots in the scheduling algorithm.

Several other efforts addressed the problem of mitigating timing covert channels. Chen and Venkataramani [2014] presented CC-Hunter—a framework for detecting the presence of covert channels by dynamically tracking conflict patterns over the use of shared processor hardware. As CC-Hunter is based on detecting contention, it is not directly applicable to detecting the covert channels through branch predictors proposed here, as these channels are not created based on contention. Another fundamental approach that builds the system from the ground up to detect the presence of side channels [Domnitser et al. 2012], covert channels, and other unintended information flows is gate-level information flow tracking (GLIFT) [Tiwari et al. 2009; Oberg et al. 2014]. Although shown to be effective, GLIFT requires significant rearchitecting and redesign of the entire system. A recently proposed technique to mitigate side channels using obfuscated execution [Rane et al. 2015] can in principle be used to also close covert channels, but its performance overhead is significant. Askarov et al. [2010] introduced a timing channel mitigation methodology that can achieve predefined bounds on the channel leakage.

Hunger et al. [2015] outlined a contention-based covert channel through a branch predictor. In this article, we quantitatively compared the channel of Hunger et al. [2015] (which we refer to as CC) with the channel based on the residual state of the branch predictor left by the trojan. We performed the comparison in both noiseless and noisy environments and demonstrated that CC is only practical in the noiseless environment, and even then it provides a signal with a lower amplitude than RSC.

Although the focus of this article is on covert channels, previous work studied side-channel attacks through branch prediction units [Aciicmez et al. 2007a, 2007b], particularly exploiting the branch target buffer. Therefore, in the future, it is important to consider mitigation techniques that will close the possibilities for both side channels and covert channels through shared branch prediction units and other shared resources. Identifying and mitigating side and covert channels becomes a high priority research direction in the environments that assume potentially compromised system software layers [McKeen et al. 2013; Evtyushkin et al. 2014; Elwell et al. 2014, 2015; Hofmann et al. 2013]. In this case, the OS can assist in the creation of the timing channels, circumventing strong isolation [Xu et al. 2015].

10. CONCLUDING REMARKS

We performed a systematic analysis and comparison of two types of covert channels through branch prediction structures - the contention-based channel (CC) and the residual state–based channel (RSC). We showed that in the clean execution environment where only the trojan and the spy processes execute, both channel are effective, with RSC providing significantly higher signal amplitude. This is true for both single-threaded and multithreaded cores. We also evaluated and compared both types of channels in an environment with the interference from one other unrelated process. Our results demonstrate that whereas RSC is still an effective channel in this situation, any level of interference becomes detrimental to the quality of CC. This is because CC is based on the presence or absence of contention for the shared branch predictor resources, but the external noise makes it impossible for the spy to observe contention-free periods. We also demonstrated that a high-capacity RSC can be created with minimal error rate: for example, a channel with about 120kbps bitrate can be constructed with only about a 4% error rate in covert communication. Finally, we proposed a software-based mitigation technique that randomizes the predictor state on every context switch and showed that the protection can be achieved with modest-performance impact.

REFERENCES

O. Aciicmez, K. Koc, and J. Seifert. 2007a. On the power of simple branch prediction analysis. In *Proceedings of the Symposium on Information, Computer, and Communication Security (ASIACCS'07)*. IEEE, Los Alamitos, CA.

O. Aciicmez, K. Koc, and J. Seifert. 2007b. Predicting secret keys via branch prediction. In *Proceedings of the Cryptographers' Track at the RSA Conference*.

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 297–307.

J. Chen and G. Venkataramani. 2014. CC-hunter: Uncovering covert timing channels on shared processor hardware. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. ACM, New York, NY, 216–228.

M. Co and K. Skadron. 2001. The effects of context switching on branch predictor performance. In *Proceedings of the 2001 IEEE International Symposium for Performance Analysis of Systems and Software.*

L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side-channel attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4, Article No. 35.

Jesse Elwell, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2014. A non-inclusive memory permissions architecture for protection against cross-layer attacks. In *Proceedings of the 2014 IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, Los Alamitos, CA.

Jesse Elwell, Ryan Riley, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Iliano Cervesato. 2015. Rethinking memory permissions for protection against cross-layer attacks. *ACM Transactions on Architecture and Code Optimization* 12, 4, Article No. 56.

Marius Evers, Po-Yung Chang, and Yale N. Patt. 1996. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. *ACM SIGARCH Computer Architecture News* 24, 3–11.

Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. 2014. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE, Los Alamitos, CA, 190–202.

Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2015. Covert channels through branch predictors: A feasibility study. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, New York, NY, 5.

Virgil D. Gligor. 1993. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*. National Computer Security Center.

Mordechai Guri, Matan Monitz, Yisroel Mirski, and Yuval Elovici. 2015. BitWhisper: Covert signaling channel between air-gapped computers using thermal manipulations. arXiv:1503.07919.

Richard W. Hamming. 1950. Error detecting and error correcting codes. *Bell System Technical Journal* 29, 2, 147–160.

O. Hofmann, S. Kim, A. Dunn, M. Lee, and E. Witchel. 2013. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 265–278.

Wei-Ming Hu. 1992. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, Los Alamitos, CA, 52–61.

Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, Los Alamitos, CA, 639–650.

Intel. 2010. Intel 64 and IA-32 Architectures Software Developer Manual. Available at http://www.intel.com

Alexey Kopytov. 2004. SysBench: A System Performance Benchmark. https://github.com/akopytov/sysbench.

Scott McFarling. 1993. *Combining Branch Predictors*. Technical Report TN-36. Digital Western Research Laboratory.

F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Svagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. Article No. 10.

J. Oberg, S. Meiklejohn, T. Sherwood, and R. Castner. 2014. Leveraging gate-level properties to identify hardware timing channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 9, 1288–1301.

Matt Ramsay, Chris Feucht, and Mikko H. Lipasti. 2003. Exploring efficient SMT branch predictor design. In *Proceedings of the Workshop on Complexity-Effective Design, in Conjunction with ISCA.*

Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. 431–446.

T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, New York, NY.

B. Saltaformaggio, D. Xu, and X. Zhang. 2013. BusMonitor: A hypervisor-based solution for memory bus covert channels. In *Proceedings of the 2013 European Workshop on System Security (EUROSEC'13).*

M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectureal Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, 109–120.

Y. Wang, A. Ferraiuolo, and E. Suh. 2014a. Timing channel protection for a shared memory controller. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, Los Alamitos, CA.

Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. 2014b. Timing channel protection for a shared memory controller. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, Los Alamitos, CA, 225–236.

Z. Wang and R. Lee. 2006. Covert and side channels due to processor architecture. In *Proceedings of the Annual Computer Security Applications Conference*. IEEE, Los Alamitos, CA.

John C. Wray. 1991. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, Los Alamitos, CA, 2–7.

Z. Wu and H. Wang. 2012. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*. 9.

Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 36th IEEE Symposium on Security and Privacy (S&P'15)*. 640–656.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (S&P'09)*. IEEE, Los Alamitos, CA, 79–93.

Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 32nd 2011 IEEE Symposium on Security and Privacy (S&P'11)*. 313–328.

Received October 2015; revised December 2015; accepted December 2015