Covert Channels Through Branch Predictors: A Feasibility Study

Dmitry Evtyushkin
CS Department
State University of New York
at Binghamton
devtyushkin@cs.binghamton.edu

Dmitry Ponomarev
CS Department
State University of New York
at Binghamton
dima@cs.binghamton.edu

Nael Abu-Ghazaleh
CSE and ECE Departments
University of California
Riverside
naelag@ucr.edu

ABSTRACT

Covert channels through shared processor resources provide secret communication between malicious processes. In this paper, we introduce a new mechanism for covert communication using the processor branch prediction unit. Specifically, we demonstrate how a trojan and a spy can manipulate the branch prediction tables in a way that creates high-capacity, robust and noise-resilient covert channel. We demonstrate this covert channel on a real hardware platform both in Simultaneous Multi-Threading (SMT) and single-threaded settings. We also discuss techniques for improving the channel quality and outline possible defenses to protect against this covert channel.

1. INTRODUCTION

Modern computer systems are typically shared by multiple applications which often belong to different security domains. Therefore, to provide security, system software layers often impose restrictions on the use of hardware resources. For example, the Android mobile Operating System (OS) requires users to explicitly grant permissions for each application. Specifically, some applications can be granted access to the network, while others can be restricted from performing outside communication, but still can read sensitive user data. To illustrate this 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 over the network to prevent password leakage. While the password manager application can by itself be buggy, or even contain embedded backdoors, the user passwords will remain secure as long as the OS correctly enforces network access permissions. At the same time, it is essential for the weather widget to have network access enabled to properly support its functionality.

This situation 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 that is restricted by the OS? One possible way to achieve this goal is to utilize a covert communication channel through processor hardware resources that are shared by both applications. In particular, a covert channel can be created between a trojan process and a spy process running on the same processor; to transmit sensitive information, the trojan alters the state of a shared hardware resource in order to intentionally modulate events on that resource. On the receiving side, the spy performs measurements to determine how the trojan is accessing the resource allowing it to receive the modulated events. We present our threat model and assumptions in Section 2.

In this paper, we present a covert channel through branch predictor that is based on two observations.

- **The state of a branch predictor is shared by all processes executing on the same core**. When a program executes a large block of *taken* or *non-taken* branches, the predictor structures are put in the corresponding state. The next time this program executes the same block of branches with the same outcomes, its branch misprediction rate will decrease significantly, leading to shorter execution time. In other words, the branch prediction unit can be controlled to reduce the execution time of blocks containing branch instructions with appropriately configured outcomes.
- **The state of a branch predictor is preserved across context switches**. When a context switch occurs, the internal structures of the branch prediction unit do not get invalidated. Thus, they can affect the branch prediction accuracy of the newly scheduled process (a spy). In an SMT environment where the spy executes concurrently with the trojan on the same core, the second property is not critical for the attack, as the spy can probe the shared branch predictor while the trojan sets its state.

Using these observations, we implement covert channel in the following way. When the trojan intends to send a *one* to the spy, it executes a large block of conditional branch instructions with *taken* outcomes). At the same time, the spy continuously executes a large block of *taken* branch instructions and measures its execution time. When the execution time is below average due to the lower branch misprediction rate (because the trojan put the predictor to the taken state), the spy detects the transmission of *one*. Transferring the value of *zero* between the trojan and the spy is achieved in the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
HASP '15 June 13 2015, Portland, OR, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ACM 978-1-4503-3483-9/15/06$15.00
DOI: http://dx.doi.org/10.1145/2768566.2768571.

same way, with the trojan intentionally executing *not-taken* branches. The spy executes the same code block of *taken* branches, thus experiencing a larger number of branch misprediction. Consequently, spy's execution time for this code block increases, allowing it to infer the transmission of *zero*. We present the channel and evaluate it on a real processor in Sections 3 and 4.

Having shown the feasibility of the channel, we discuss in Section 5 possible techniques to prevent its use in practice. In particular, we propose partitioning for SMT settings, and BPT flushing on a context switch. Our future work will explore the security properties and performance impacts of these suggested solutions.

Our work contributes a new covert channel through the branch prediction unit. Although other covert channels have been demonstrated using shared microarchitecture resources, to the best of our knowledge only one previous effort considered covert channel through branch predictors [12]. We compare our covert channel to the work of [12], and review other related work in Section 6.

The main contributions and the key results of this paper are:

- We demonstrate the feasibility of a covert communication channel through shared branch predictors on a real hardware platform.
- We show that covert channels through branch predictors can be created in both SMT and single-threaded settings.
- We demonstrate the resilience of this covert channel to noise from other concurrently running applications and its ability to achieve high communication bandwidth.
- We suggest possible improvements to the channel, as well as techniques to mitigate it.

2. 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, therefore 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 multi-core processor.

The system software is assumed to be uncompromised, so that it properly enforces access control and preserves legal information flow. The two processes need only 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 [26]. However, if the access to performance counters is available, than a significantly better signal quality can be achieved.

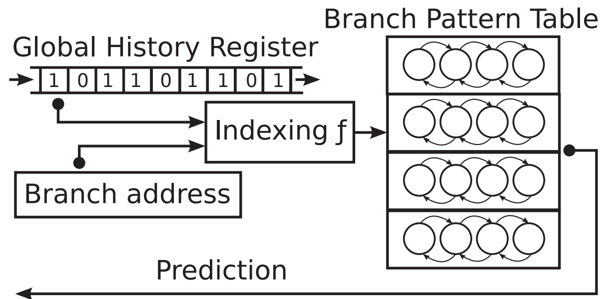


Figure 1: Schematic of a *gshare* predictor

3. COVERT CHANNEL THROUGH BRANCH PREDICTOR

In this section, we describe a generic branch predictor design and its features that make the secret data communication between the trojan and the spy possible. After that, we describe the construction of the covert channel through branch predictor in detail. We start by overviewing our experimental setup and evaluation methodology.

3.1 Evaluation Methodology and Experimental Platform

We demonstrate the covert channel through branch predictor on real hardware, rather than on a microprocessor simulator. While a simulator can provide some additional insight into the behavior of the branch predictor, it could also oversimplify important hardware features present in real chips.

All our experiments were performed on a machine with an Intel Core i7-4800MQ CPU (Haswell microarchitecture) clocked at 2GHz. The machine has 16GB of DDR3 memory clocked at 1600 MHz. We consider systems with and without SMT - to evaluate the latter we disabled the SMT support. We also consider the effect of interference from different programs. The machine uses Ubuntu 14.04.2 LTS operating system, with a generic GNU/Linux kernel version 3.16.0-31.

3.2 Dynamic Branch Predictor

Modern high-performance processors are deeply pipelined. Therefore, a significant number of cycles is needed to resolve the outcome of conditional branch instructions. To avoid stalling the pipeline, branch prediction is performed to allow the processor to speculatively execute instructions before the branch is resolved. 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 as well as overhead to undo the side effects of the erroneous speculation.

A number of different schemes for branch prediction have been proposed. They range from simple static predictors, to complicated neural-network based predictors [13]. However, most of currently available microprocessors rely on some form of *correlating two-level predictors*, such as *gshare* [14]. Two-level predictors combine the global history of recently executed branches with local history for individual branches. Such a combination allows this predictor to use different saturating counters for the same branch, depending on the control flow path leading to the execution of a branch being predicted, resulting in highly accurate

predictions.

Figure 1 presents a schematic of a gshare predictor [14]. Note that our covert channel will work with any predictor, we describe the *gshare* predictor because of its wide use and to provide a concrete example. The global history register is a shift register that accumulates the history of several recently executed branches. The Branch Pattern Table (BPT) is a relatively large table of two-bit saturating counters, with the counter values 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 global history register. Thus, the resulting indexed BPT entry is chosen based on both global and local branch information. One of the most important properties of any branch predictor is the size of the BPT. Branch predictors with larger table sizes have fewer collisions (the case when two different branches get mapped to a same table entry), thus exhibiting a better prediction accuracy.

3.3 Transmitting Data through Branch Predictor Structures

Transmitting data through the branch predictor covertly is possible due to an important design consideration that is true for most of the systems and branch predictor implementations today. Specifically, the BPT is not flushed on a context switch. Several branch predictor designs [6] have been introduced that considered context switches that erase the branch history data from the old context in the BPT. However, these designs have not been adopted in commercial products, as no performance benefits were observed [4].

Since the BPT is not flushed, the branch history from the old context remains available to the newly scheduled process. Such a situation allows the trojan application to communicate to the spy by setting the BPT into one of (at least) two pre-agreed on states. For example, the trojan can set all of the BPT entries to the *strongly taken* state to communicate a *1*. After the spy process is context-switched into the CPU, it can observe the BPT state created by the trojan by performing a series of measurements.

This general technique is also available in Simultaneously Multithreaded (SMT) processor cores. SMT cores allow the processor to fetch instructions from two independent processes simultaneously. The SMT cores share the same branch predictor hardware and its data structures among the threads. While it is possible to design a branch predictor with split data structures for the simultaneous threads, such splitting does not bring significant performance improvements [19] and thus is not typically used. In Section 4, we demonstrate the covert channel with both enabled and disabled SMT.

3.4 Building the Covert Channel

In order to demonstrate the feasibility of covert channels through branch predictor, we execute two programs on the same physical microprocessor core, the trojan and the spy. The purpose of the trojan is to fill the branch predictor's BPT in such a way that it affects the time of execution of the code blocks enough to distinguish it from the noise introduced by the Operating System or other programs that are scheduled to execute on the same core.

In particular, the trojan executes a block of exclusively *taken* conditional branch instructions to transmit *one*, and a block of *not-taken* branch instructions to transmit *zero*. The spy process, when it receives a time slice from the operating system to execute on a CPU, probes the context of the BPT by executing a block of taken branch instructions and measuring the execution time. If the spy process experiences performance slowdown, this indicates mispredictions implying that the trojan left the predictor states as *not-taken* intending to transmit a *zero*. Alternatively, when the block of taken branches executes quickly, the transmission of *one* is determined. Note that the spy measures only the execution time (or alternatively branch mispredictions through performance counters) of the starting period of its execution slices, which is impacted by the residual information left in the BPT by the trojan.

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

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

(a) Main loops of the trojan and the spy processes

```
br_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
......................
```

(b) Taken code block

```
br_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
```

(c) Not-taken code block

Figure 2: Code used to fill and probe BPT

Figure 2 presents an example trojan and spy code that can be used for filling and probing the BPT respectively. Figure 2a shows the main loops of both programs. The trojan code executes an infinite loop, alternating the transmission of *zeroes* and *ones*. During odd seconds, it primes the BPT with *taken* predictions (in order to transmit *one*), and during even seconds, it primes the BPT with *non-taken* predictions to transmit *zeroes*.

Figures 2b and 2c present the code blocks used to fill the BPT of the branch predictor with *taken* and *not-taken* predictions respectively. Both figures show the pattern that is used to deploy much larger blocks. We observed that the best covert channel quality is achieved when we use larger code blocks for the trojan and smaller ones for the spy.

The goal of the trojan is to fill as many BPT entries with desired branch prediction data as possible. A large code block (several hundred thousand branches) of branch instructions is used for this purpose. Note that a short loop with a few branches that executes for many iterations cannot be used, because the branch outcomes will be written repeatedly into a few BPT entries.

The act of probing the BPT by the spy uses a similar *taken* branch code block as that used by the trojan, although the block size may be different. Another difference is that the spy's code is not executed constantly. Instead, it is only executed once for each probing time, recording the timestamp counter data. When the spy completes the execution of the block, it relinquishes the rest of its CPU timeslice. We use the `sleep()` function

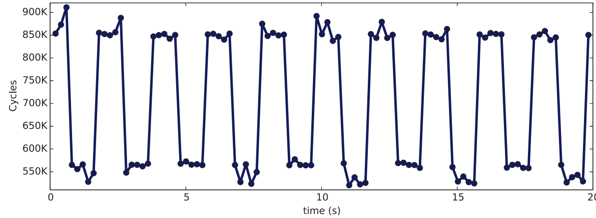


Figure 3: Timing of the branch code block in the spy process, reflecting the data sent by the trojan.

for this purpose. The duration of the executed block is chosen to ensure that the branch predictions performed within that block are affected by the residual state from the trojan, and not by the history that is already built by the spy during the current slice.

We inserted a uniformly distributed number of `nop` instructions (from 2 to 8) between the conditional branches in both trojan and spy processes. Primarily, we used this approach to improve the chance of filling in more entries in the BPT. Since our experiments are performed on real hardware, we have no knowledge of the exact details of the BPT indexing function. We observe that adding the `nops` increases difference between the *ones* and *zeroes* measured by the spy, indicating an improved quality of the covert channel signal. If we invest in reverse engineering the branch predictor hashing function, we can more efficiently prime the BPT to maximize the bandwidth of the channel, and even communicate multiple bits per prime-probe cycle.

Figure 3 presents our initial results from the experiment when we executed the trojan and the spy processes in a manner described above. The x-axis represents the time (in seconds) from the moment the spy program starts probing the BPT. The y-axis represents the number of cycles spent executing the branch code block. For this particular experiment, the trojan executes *500* thousand branch instructions and the spy executes *30* thousand branches. The graph shows a clear separation between the two states being communicated, demonstrating the feasibility of the channel.

3.5 Covert Channels in Single-Threaded and SMT Modes

In most microprocessors, each core is equipped with its own branch predictor and its own BPT. It is therefore reasonable to assume that the data transmission mechanism described above will only work in the case when the trojan and the spy are executed on the same physical core. We confirmed this with our experiments - no covert channel was observed when the trojan and the spy were running on different cores. However, in some settings this may not be a significant limitation, because the operating system can allow user applications to set process's affinity, as in GNU/Linux [18].

A single physical core can either run in single-thread mode, or in Simultaneous Multithreaded (SMT) mode. When the core executes in the single-thread mode, it fetches the instructions only from one process at a time. As we pointed out earlier in this section, the branch predictor hardware is usually shared among different programs running on the same core sequentially. Branch predictor's data structures are not flushed or invalidated on context switches. Note, that before actual context switch happens, some operating system code must be executed. However, we discovered that this code does not cause significant BPT pollution as demonstrated in our results so far, which include these effects.

For a single-threaded core setting, it is very important to achieve consecutive scheduling of the trojan and the spy. Otherwise, if another program gets executed between them, this program will populate the BPT with its own branch information, removing data put by the trojan. In this case, there is a high probability that the spy would not be able to extract the data from the BPT. For our experiments we achieve the consecutive scheduling by dedicating a CPU core only to these two programs, using the default operating system functionality. Since the trojan is constantly running its block of branches and the spy runs its block just once before forcing the context switch, we obtain an ideal scheduling. In particular, the spy interposes the trojan's execution, probes the BPT and switches back to the trojan.

In case when the SMT is enabled, the trojan and the spy do not need to achieve the strict consecutive scheduling. In contrast, they need to achieve simultaneous scheduling, when they both are running on the same physical core, but using different hardware thread contexts. For our experiments under the SMT conditions, we assign both processes to isolated virtual cores (a single SMT-enabled physical core is represented in the operating system as two virtual cores). In this case, the trojan and the spy execute on the same physical, but on different virtual cores.

The presence or the absence of SMT in the system changes the way the information is transmitted from one process to another. However, the change in the source code of both programs is not required. A detailed comparison of SMT and non-SMT covert channels is presented in Section 4.2.

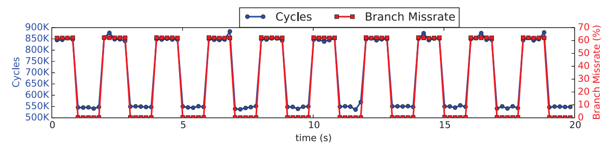
4. RESULTS AND DISCUSSION

In this section, we present the results of our experiments obtained by executing the spy and the trojan programs on the hardware platform as described in Section 3.4. First, in order to confirm the expected behavior of the branch predictor, we collected low-level information on the number of executed branch instructions and the number of branch mispredictions, instead of relying solely on the timestamp counter. Then, we examine the behavior of our covert channel under various systems configurations, focusing on the resilience to noise and consistency of the timing channel. We also analyze the properties of our covert channel both in single-threaded and SMT execution environments.

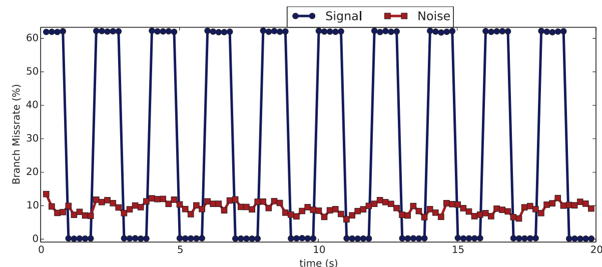
4.1 Branch Mispredictions and Execution Time

In our earlier experiment (shown in Figure 3) we demonstrated the behavior of the spy process under the influence of the trojan. When the trojan executes branch instructions that have the opposite outcomes compared to the spy, the spy experiences a slowdown. On the other hand, when the trojan executes branches with the same outcome as the spy, the spy experiences a speedup. These effects were observed using the timestamp counter. However, the value of the timestamp counter does not provide any direct insight into the sources of performance gain or slowdown.

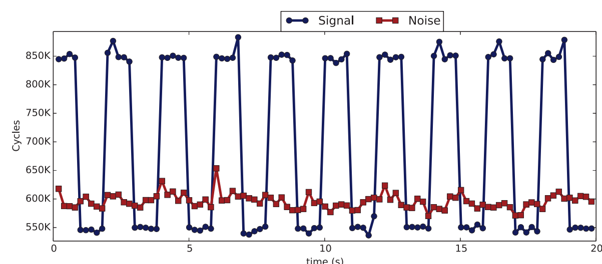
To get a better insight into the reasons for the performance differences, we integrated the functionality for reading the performance counters within the spy pro-



(a) Execution time dependency on the misprediction rate in the spy



(b) Signal-to-noise comparison in measured misprediction rate



(c) Signal-to-noise comparison in measured execution time

Figure 4: Spy's measurements demonstrating trojan's impact on the branch prediction rate and execution time.

cess. We then repeated the same experiment and obtained both the timestamp counter data and the branch misprediction rate. We recorded the total number of branch instructions and the number of branch mispredictions.

For comparison purposes, we also evaluated the system where the spy process runs alongside a program that exercises the branch predictor in a random manner without creating special conditions for the spy. We used the `cpuburn` [16] benchmark for this purpose. The `cpuburn` benchmark serves as a source of branches with realistic outcomes to fill the BPT with data that is representative of normal program behavior.

The results of these experiments are presented in Figure 4. Figure 4a demonstrates the changing patterns in both the branch misprediction rate and the number of execution cycles. Figure 4b shows the comparison of the covert channel signal to the normal noise, measured in terms of branch misprediction rate. Finally, Figure 4c demonstrates the comparison of the signal to normal noise levels, measured in cycles.

The data presented in Figure 4a reflects the dependency of the number of cycles that the spy process spends running the branch code block (measured in processor cycles) on the number of branch mispredictions. It is clear from these graphs that the execution time is dependent on the misprediction rate. As expected, a higher misprediction rate results in higher execution time.

Instead of measuring its execution time, the spy can use branch-related performance counters as a measurement mechanism. In fact, relying on the performance counters provides higher measurement accuracy. The covert channel signal measured with branch-related performance counters is consistent and has excellent signal-to-noise ratio, as we demonstrate later in this section. However, reading performance counters may require administrative privileges from the spy. In fact, whether such privileges are required or not depends on the particular hardware, operating system and even hardware configuration. For example, according to the Intel's Architecture Software Developers Manual [10], a particular configuration set allows or disallows user-level accesses to the performance counters. However, we conservatively assume that the performance counters are not always available and target the timestamp counter as our main measurement mechanism.

Based on the collected data, we make several observations:

Observation One.

The trojan's activity significantly affects the measured branch misprediction rate of the spy process. When the trojan is executing branch instructions with different outcomes than spy's, the branch misprediction rate of the spy process significantly increases. In particular, the mean value of the high peaks is *62.06%*. At the same time, when the trojan executes the branches with the same outcome as those executed by the spy, the misprediction rate decreases to very low values - the mean value of the low peaks is just *0.21%*. The implication of this statistics is that almost all of the branches are correctly predicted. Such a high ratio between the low and the high values (nearly *300x*) contributes to the noise resilience and capacity of the covert channel.

Observation Two.

Although the number of cycles required to execute the spy precisely repeats the pattern of the high and low peaks measured in terms of the branch misprediction rate, the difference between high and low peaks tends to be less significant, as expected. In particular, the mean high peak is *852748.02* cycles, and the mean low-peak is *547929.72* cycles. The ratio between these two values is only *1.56x*, which is much smaller than what we observed when the misprediction rate was measured directly, perhaps indicating a smaller noise margin. Nevertheless, this difference is substantial, allowing clear distinction between the bits being transmitted.

Observation Three.

The signal depicted on Figures 4b and 4c are highly consistent: the signal values have a little deviation across different bits of the same value. This is also true for the values of the noise signal. To quantify the consistency of the signal, we calculated the coefficient **F**. We define it as:

$$F_{max} = \frac{\sigma_{max}}{|\mu_{max} - \mu_{min}|}, \; F_{min} = \frac{\sigma_{min}}{|\mu_{max} - \mu_{min}|} \quad (1)$$

Where σ_{max} and σ_{min} is the standard deviation for the high and the low peaks and μ_{max} and μ_{min} are the mean values of the high peaks and the low peaks respectively.

Intuitively, this value shows how spread out the measured values in the peaks are, compared to the range of the signal. The smaller value indicates a more consis-

tent measurement. The **F** values of the high and the low peaks of the branch misprediction rate are *0.22%* and *0.08%* respectively. For the cycles measurement, the value of **F** is *5.16%* and *1.65%* for the high and the low peaks, respectively.

Observation Four.
We observed that the noise level stays between the two peaks of the amplitude of the signal. It allows the spy to utilize both peaks of the amplitude for the signal transmission (for transmitting either *0* or *1*). At the same time, since it is easy for the spy to distinguish the noise from the transmitted signal, asynchronous transmission can be used. This is a very desirable property for a covert channel, since the spy and the trojan do not have a communication channel to arrange the actual transmission time.

Observation Five.
In order to assess the relationship between the signal and the noise, we propose the coefficient **G** defined as,

$$G = Max\left(\frac{|\mu_{max} - \mu_{noise}|}{\mu_{noise}}, \frac{|\mu_{min} - \mu_{noise}|}{\mu_{noise}}\right) \quad (2)$$

where μ_{max} is the mean of the maximum peaks, μ_{min} is the mean of the minimum peaks, and μ_{noise} is the mean of the noise. This coefficient shows how easy it is for the spy to differentiate the signal from the natural noise level. This coefficient was computed using only one of the two peaks to reflect maximum possible value of the coefficient. Larger values of **G** imply better results. The value of **G** for the experiment shown in Figure 4 is *5.34* for the branch misprediction rate, and *0.42* when measured in cycles.

Observation Six.
Based on the collected data (execution cycles and the number of mispredictions) we calculated the cost of the branch misprediction. We observed the misprediction cost is within the range between *10* and *18* cycles. Similar numbers have been reported for Haswell CPUs in [8].

In summary, the presented characteristics of the covert channel demonstrate that the covert channel through branch predictor is consistent, noise resilient, and has a high capacity.

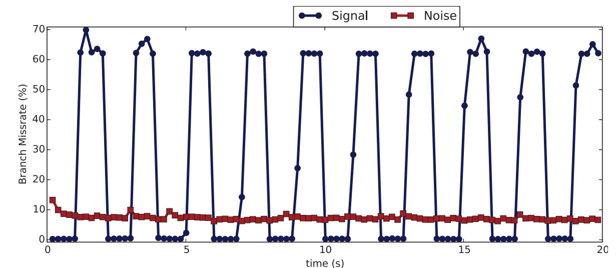
4.2 Covert Channel in the SMT Mode

For the second experiment, we executed the spy and the trojan on the same physical core, but on two different virtual cores (hardware contexts) in SMT mode. The results are presented in Figure 5: Figure 5a uses branch misprediction rates while Figure 5b shows cycles.

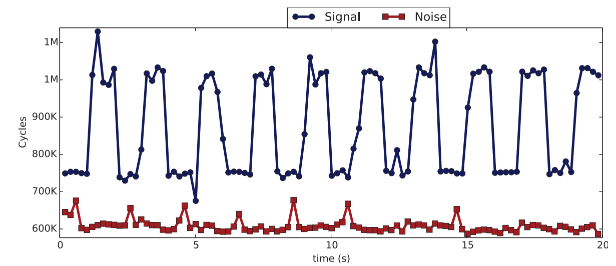
These results show that the branch predictor covert channel exists in SMT mode as well, but with a slightly decreased consistency. The noise level shows the same pattern and levels.

At the same time, the results of cycle measurements generally repeat the pattern exhibited in single-threaded mode. However, all results impacted by the trojan now rise above the noise level. Such increase in the number of cycles can be attributed to effects such as competition for the instruction caches. In the SMT mode, the two threads are executing at the same time. The trojan has a relatively large code size and is constantly executing in a loop. As a result, the spy executing in parallel has to compete for the cache capacity (and other shared CPU resources) with the trojan.

The covert channel on SMT exhibits attractive characteristics. The value of **F** coefficient is *7.6%* and *8.67%* for the high and the low peaks respectively when branch misprediction rate is measured. These values are *12.62%* and *11.88%* when the the execution cycles are measured. The value of **G** coefficient is *7.43* and *0.67* for the branch misprediction rate and execution cycles respectively.



(a) Signal-to-noise comparison in misprediction rate



(b) Signal-to-noise comparison in execution time

Figure 5: Covert channel characteristics when the spy and the trojan run on different virtual cores in SMT mode.

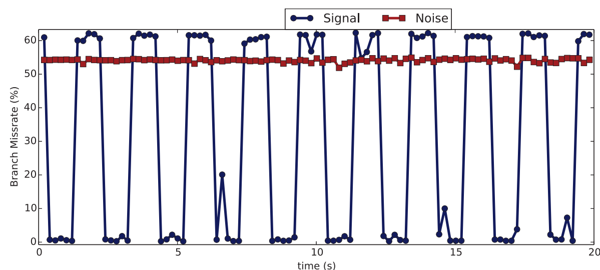
4.3 Covert Channel under Realistic Conditions

Finally, in order to assess the applicability of the covert channel on systems with realistic workloads, we executed the trojan and the spy on a single core with the Firefox browser, playing a YouTube video in the background. We ran this experiment under single-thread configuration. The results of this experiment are presented on Figure 6. The branch misprediction rate is shown on Figure 6a, and execution cycles are presented presented in Figure 6b.

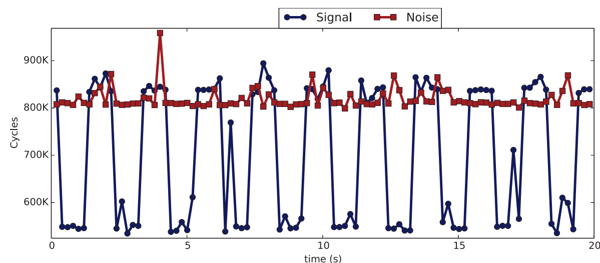
Note that in all prior experiments we used the *cpuburn* benchmark in order to obtain the nominal noise level data. The purpose of running this benchmark in the background was to remove the data from the BPT and execute the spy under a normal state of BPT. We do not need to run such benchmark for this experiments, because the browser is running on the same core and plays the role of the noise benchmark in this experiment. The browser was executed during all stages of the experiment, including filling the BPT by the trojan, probing it by the spy, and capturing the noise.

The **F** coefficient of the covert channel under this scenario was measured as *2.47%* and *5.29%* for the high and the low peaks for the branch misprediction rate measurements. The value of this coefficient was *9.53%* and *6.56%* when the execution cycles were measured. The value of **G** coefficient was measured as *0.97* and *0.32* for the branch misprediction rate and execution

cycles respectively. Thus, the covert channel continues to be effective.



(a) Signal-to-noise comparison in misprediction rate



(b) Signal-to-noise comparison in execution time

Figure 6: Covert channel under realistic load. The Firefox browser is executed on the core while tojan transmits the signal.

5. MITIGATING BRANCH PREDICTOR BASED COVERT CHANNELS

While detailed analysis of protection and mitigation techniques is beyond the scope of this paper, we outline some general protection approaches in this section. The following techniques can be considered to mitigate the covert channels created through shared branch predictors. First, it is possible to save the branch predictor state of each process and restore it on every context switch. This way, one process cannot interfere with the predictor state used by another process. However, this will add some slowdown to the context switching process. Another alternative is to simply flush the BPT on every context switch, but that requires additional hardware and may introduce extra delays. Note that this technique can only protect against single-thread covert channel.

In order to mitigate the covert channel in SMT setting, the BPT can be physically partitioned between multiple thread contexts (thus achieving isolation), but this can have negative impact on branch accuracy and performance. Finally, unusual anomalies such as all-taken branches can be detected by hardware and such programs can be terminated, similar to [9]. However, it is difficult to correctly detect all anomalies, since the trojan can intentionally introduce some randomness into its activity to evade the detection.

6. RELATED WORK

Covert channels through shared microprocessor resources have been explored in several recent efforts. Wang and Lee [24] presented covert channels using exceptions on speculative load instructions and shared functional units on SMT processors. Wu et al. [25] describe a covert channel that is based on the Intel Quick Path Interconnect (QPI) lock mechanism. Ristenpart et al. [20] present a cross-VM covert channel that exploits the shared cache. Covert channels based on the use of memory bus were presented in [21]. Wang et al. [23] 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.

A number of other efforts addressed the problem of mitigating timing covert channels. In [3], Chen and Venkataramani present 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, because 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 [5], covert channels, and other unintended information flows is GLIFT (Gate-level information flow tracking) [22, 17]. While shown to be effective, GLIFT requires significant rearchitecting and redesign of the entire system.

Hunger et al. [12] outlined a covert channel through branch predictor that is constructed in a different way from ours. In particular, the trojan in [12] transmits a *one* by executing a large number of branches taken with 50% probability, and it transmits a *zero* by busy-waiting. The trojan's activity during the transmission of a *one* creates the branch predictor resource contention, which is detected by the spy using performance counters. In contrast, our covert channel does not rely on branch predictor contention, but uses residual state of the branch pattern table (BPT) that is left from the Trojan after a context switch. In addition, we demonstrated that our covert channel can be constructed without relying on performance counters by simply monitoring the execution time of the spy. Finally, we studied the branch predictor covert channel under various settings with realistic system workloads.

While the focus of this paper is on covert channels, previous work studied side-channel attacks through branch prediction units [1, 2]. 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. This is an important problem for both traditional environments (with OS and VM based isoalation), and systems that support stronger isolation guarantees [15, 7, 11].

7. CONCLUDING REMARKS

We introduced a new covert channel that uses dynamic branch predictor structures to perform secret communication between the trojan and the spy processes. The key idea is that the trojan process can fill in the predictor structures by *taken* or *non-taken* predictions, thus impacting the branch prediction accuracy and the execution time of the spy process as it executes a block of code composed of taken branches. In order to detect timing differences, the spy process can either measure its total execution time (using a timestamp counter), or can read the performance counters to di-

rectly observe the branch misprediction rate (provided that is has permissions to do so). We demonstrated that a practical covert channel is possible in both cases. In addition, we show that the channel has high capacity and is resilient to interference from external processes.

8. ACKNOWLEDGEMENT

This material is based on research sponsored by the National Science Foundation grant CNS-1422401.

9. REFERENCES

[1] Aciicmez, O., Koc, K., and Seifert, J. On the power of simple branch prediction analysis. In *Symposium on Information, Computer and Communication Security (ASIACCS)* (2007), IEEE.

[2] Aciicmez, O., Koc, K., and Seifert, J. Predicting secret keys via branch prediction. In *The cryptographers' track at the RSA conference* (2007).

[3] Chen, J., and Venkataramani, G. Cc-hunter: uncovering covert timing channels on shared processor hardware. In *MICRO* (2014), ACM.

[4] Co, M., and Skadron, K. The effects of context switching on branch predictor performance. *2001 IEEE International Symposium for Performance Analysis of Systems and Software* (nov 2001).

[5] Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., and Ponomarev, D. Non-monopolizable caches: Low-complexity mitigation of cache side-channel attacks. In *ACM Transactions on Architecture and Code Optimization, Special Issue on High Performance and Embedded Architectures and Compilers* (Jan. 2012).

[6] Evers, M., Chang, P.-Y., and Patt, Y. N. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ACM SIGARCH Computer Architecture News* (1996), vol. 24, ACM, pp. 3–11.

[7] Evtyushkin, D., Elwell, J., Ozsoy, M., Ponomarev, D., Ghazaleh, N. A., and Riley, R. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on* (2014), IEEE, pp. 190–202.

[8] Fog, A. The microarchitecture of intel, amd and via cpus. *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering* (2014).

[9] Gianvecchio, S., and Wang, H. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 307–316.

[10] Guide, P. Intel® 64 and ia-32 architectures software developer's manual.

[11] Hofmann, O., Kim, S., Dunn, A., Lee, M., and Witchel, E. Inktag: Secure applications on an untrusted operating system. In *Proceedings of ASPLOS* (2013).

[12] Hunger, C., Kazdagli, M., Rawat, A., Dimakis, A., Vishwanath, S., and Tiwari, M. Understanding contention-based channels and using them for defense. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 639–650.

[13] Jiménez, D. A. Fast path-based neural branch prediction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (2003), IEEE, pp. 243–252.

[14] McFarling, S. Combining branch predictors. Tech. rep., Technical Report TN-36, Digital Western Research Laboratory, 1993.

[15] McKeen, F., Alexandrovich, I., Berenzon, A., C.Rozas, Shafi, H., Shanbhogue, V., and Svagaonkar, U. Innovative instructions and software model for isolated execution. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13* (2013).

[16] Nielsen, P. cpuburn Website. `https://patrickmn.com/projects/cpuburn/`. Accessed: 03-29-2015.

[17] Oberg, J., Meiklejohn, S., Sherwood, T., and Castner, R. Leveraging gate-level properties to identify hardware timing channels. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2014), IEEE.

[18] Page, L. M. sched_setaffinity (), 2006.

[19] Ramsay, M., Feucht, C., and Lipasti, M. H. Exploring efficient smt branch predictor design. In *Workshop on Complexity-Effective Design, in conjunction with ISCA* (2003), vol. 26, Citeseer.

[20] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security* (2009), ACM.

[21] Saltaformaggio, B., Xu, D., and Zhang, X. Busmonitor: a hypervisor-based solution for memory bus covert channels. In *EUROSEC Conference* (2013).

[22] Tiwari, M., Wassel, H., Mazloom, B., Mysore, S., Chong, F., and Sherwood, T. Complete information flow tracking from the gates up. In *Architectureal Supportfor Programmimng Languages and Operating Systems* (2009), ACM.

[23] Wang, Y., Ferraiuolo, A., and Suh, E. Timing channel protection for a shared memory controller. In *International Symposium on High Performance Computer Architecture* (2014), IEEE.

[24] Wang, Z., and Lee, R. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference* (2006), IEEE.

[25] Wu, Z., and Wang, H. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *USENIX Security Symposium* (2012), USENIX.

[26] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proc. 2011 IEEE Symposium on Security and Privacy (S&P)* (2011), pp. 313–328.