Branch Targeted Buffer

A Covert Channel Over BTB

Russell Martin
Dept. of Computer Science
College of William and Mary
Williamsburg, Virginia
rmmartin02@email.wm.edu

ABSTRACT

Branch prediction units (BPU) are essential to modern CPUs. They have grown increasingly complex and efficient as manufacturers look for ways to increase CPU performance. As is usual in pursuit of performance gains security implications have been overlooked. Branch prediction units have been the target of numerous covert and side channels lately, most notably Spectre and Meltdown attacks. Many attacks have focused on the branch history table of the BPU to create patterns of mispredictions. In this paper I examine a less commonly attacked component of the BPU, the branch target buffer (BTB).

My attack seeks to create a covert channel between processes via the BTB. To do this a spy process is executed with some small number of unconditional jumps. A trojan then executes with either a large number of jumps, in an attempt to evict the spy's BTB entries, or nops to transmit a 1 or 0 respectively. The spy process is then executed again and if mispredictions occur because of evicted BTB entries a timing difference will appear. This covert channel is demonstrated on a clean system with success and along with noise to some success.

1. INTRODUCTION

Efficiency and performance in CPUs has for quite a while now not been solely dependent on their clock speed or manufacturing process. All sorts of optimizations take place on today's CPUs. Caches are one of the best examples of this, using their fast access speed to better feed data to a hungry CPU. This micro-architectural enhancement has been the site of many recent attacks and continues to face threats however. Another area that is seeing a growing amount of attacks is the Branch Prediction Unit of CPUs.

Even though the program you might run appears to execute serially through its instructions the truth is far from that. Modern CPUs are given as many instructions as it is possible to execute at a certain time, committing them to processes once it is sure it has done the correct thing. The hiccup in this comes when a CPU runs into a branch instruction, it is then unsure where to continue executing instructions. If it waits for all the dependencies of the branch to execute it could lose several hundred cycles of execution time. Thus the CPU guesses the direction of the branch using a branch prediction unit. These branch prediction units (BPU) have grown quite complex with the ability to detect conditional jumps, unconditional jumps, indirect jumps, and even loops.

The branch target buffer (BTB) is a cache-like structure that holds branches and their target addresses. Many different schemes have been devised on how to do this and they vary from micro-architecture to micro-architecture. Companies don't usually share this information however because it is highly important to the performance and competitiveness of their chips. BTBs are commonly divided into sets and ways, much like caches. The amount of sets and ways varies widely but can be reconstructed using micro-benchmarks[8]. For the Haswell chip I'm experimenting on it has proven difficult to decipher the set addressing scheme, which most probably uses some sort of hashing. Evtyushkin et al.[6] however showed that only the lower 30 bits are used for addressing, with bits 31 through 48 not seeming to be used.

The vulnerability in branch prediction units comes from the fact that they don't flush their information on context switches and in machines with simultaneous multithreading (SMT) two virtual cores share a BPU. This is done for reasons of performance and/or simplicity. This however exposes the BPU

state set by one process to another process executing on that core, creating a channel through which information can be transmitted between processes of different security domains. If the targeted process is controlled by an attacker it is called a trojan process which communicates to a spy process, this type of attack is called a covert channel. If the targeted process is not controlled by the attacker it is called a victim, this is called a side channel.

2. THREAT MODEL AND ASSUMPTIONS

We assume there are two programs running, the trojan and the spy, which are either malicious or compromised. These two programs have no other communication channels and one is attempting to send the other information across security domains. It is assumed the trojan and spy are colocated on the same physical core, either on the same virtual core or on different virtual cores via SMT (BPUs are not shared across physical cores). The system software is uncompromised and all control flow and access of information is happening as intended. It is assumed that timing methods available to programs are sufficient to differentiate when the spy mispredicts branches.

3. COVERT CHANNEL THROUGH BRANCH TARGET BUFFER

In Aciicmez et al.[2] they describe four different attacks for extracting branch timing information of a target branch in a cypher algorithm. The second and third attack consists of executing the cypher, measuring the time to execute. Then clearing the BTB and measuring the cypher time again. If the victim takes longer than a misprediction occurred because of clearing the BTB. They use this to leak information about the secret bit at that point in the cypher execution.

In Evtyushkin et al. [4] they classify covert channels through branch predictors into contention based and residual based. In the contention based channel, originally described in a paper by Hunger et al. [7], a series of 50% taken and 50% not taken branches were executed to transmit a 1, and nops executed to transmit a 0. A spy then executes code with a similar 50/50 split of branches. If the spy takes longer to execute that is because it is fighting for space with the trojan, causing mispredictions. In the residual based side channel the trojan executes a large number of taken branches for 1, and a large number of not-taken branches for 0. The spy then executes a small number of taken branches. If the trojan is transmitting a 1 the spy will predict taken still and execute quickly, if 0 then the spy will mispredict its branches as not taken and take longer to execute. They found the residual channel to be more robust than the contention based channel.

To create my covert channel I sought a way to create contention in the BTB. In order to transmit data we must be able to create a difference in the BTB that can be measured by a spy process. To do this I first measure the time it takes to execute a spy process, then change the BTB with the trojan, then measure the execution time of the spy process again. The spy process consists of some smallish number of unconditional jumps. The trojan process consists of thousands of unconditional jumps in an attempt to evict BTB entries the spy process created. If transmitting a 1 the trojan will execute these unconditional branches, if not it will execute a similar amount of nop instructions. When the spy process is executed again it will experience a slowdown if its BTB entries were cleared by the trojan, indicating a 1. Code for the trojan and spy are found in listing 1 below.

```
Trojan():
        jmp trojan1
        nop
        nop
        trojan1:
        jmp trojan2
        nop
        nop
        trojan2:
        jmp trojan3
        nop
        ...
```

```
Spy():
        jmp spy1
        nop
        spy1:
        jmp spy2
        nop
        spy2:
        jump spy3
        nop
        ...
```

Listing 1. Code for trojan and spy.

3.1 Platform

For these experiments a laptop equipped with a Intel i7-4720HQ CPU at 2.6GHz with 12GB of memory at 1600 MT/s was used. The OS used was Ubuntu 18.04.1, linux kernel 4.15.0-42-generic. Experiments were conducted with SMT enabled and

disabled. This represents a real world target platform. Experiments were also conducted with noise from Mozilla Firefox playing a youtube video on the same core.

3.1 Parameters

There are four key parameters to be adjusted based on the particular micro-architecture. The size of both the trojan jumps and spy jumps should be adjusted so that successive jumps fall in different BTB sets to optimally fill the BTB and avoid landing in the same set. They should also be different in the unlikely event that the trojan and spy have overlapping virtual addresses being mapped to the BTB. That way if a trojan BTB address collides with a victim BTB address it will still evict it because the target will be wrong. Thirdly the number of branches in the trojan should be enough to fully evict the BTB, or at least to evict most of the spy's entries. Finally the number of branches in the spy should be large enough that a misprediction will slow the execution down, but not so many that it has branches mapped to the same entry causing mispredictions by itself. To choose the best combination I first varied the number of jumps in trojan with the number of spy branches set to 16, then varied the number of branches in the spy, all while keeping the trojan jump size at 5 bytes between jumps and the spy jump size at 3 bytes between jumps.

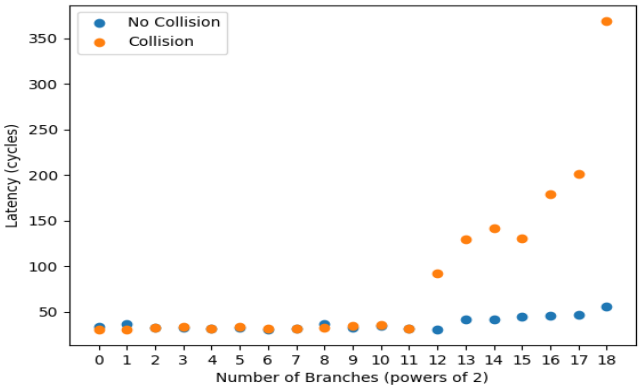

Figure 1. Varying number of branches in the trojan with 16 spy branches checking latency.

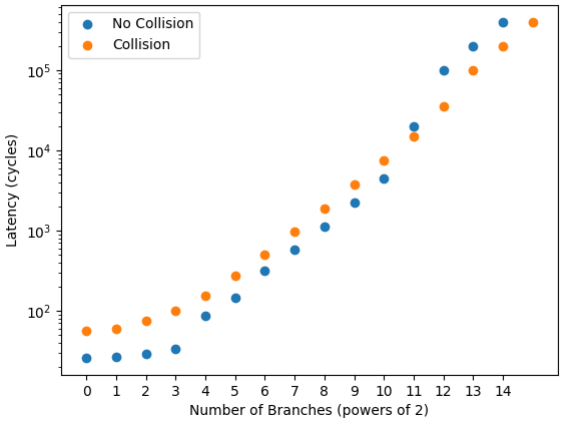

Figure 2. Varying number of branches in the spy with 2^16 branches in the trojan.

From these tests I determined to use 2^16 branches in my trojan to clear out BTB entries, though this could probably be lower with adjustments to the trojan jump size and careful examination of BTE addressing. For the number of spy branches I chose 2^4 because this gave the clearest side channel while not taking too long to execute.

3.2 Testing

Next I moved on to actually testing my covert channel transmitting data. I had the trojan change from transmitting a 1 to transmitting a 0 every second. The spy took 20 measurements every second for 20 seconds. Figure 3 shows the results of executing the spy and the trojan on the same virtual core, which means the trojan and spy are each fully executing, time sharing the CPU.

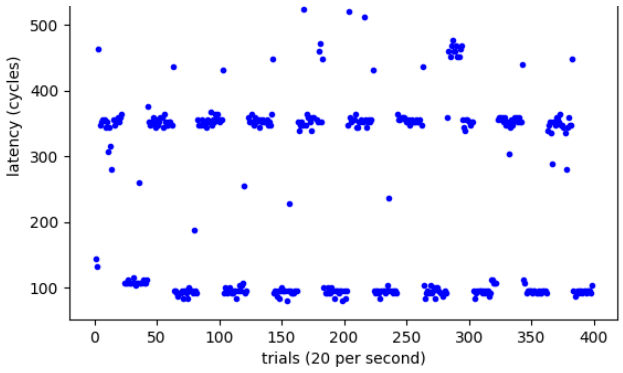

Figure 3. Spy and Trojan operating on the same virtual core.

Figure 4 is the trojan and spy executing on different virtual cores on the same physical core, this means their execution are occurring simultaneously with instructions of each being interwoven.

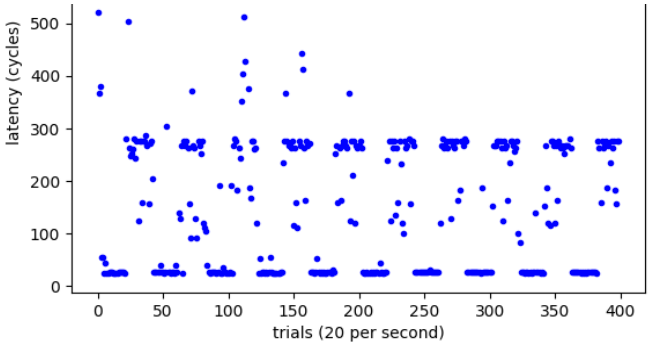

Figure 4. Spy and Trojan operating on different virtual cores on the same core.

The 1 and 0s for both are quite clear, the covert channel is working in both cases. Though the covert channel across SMT seems not quite as clear as the channel on the same virtual core, this is expected however because the trojan might not have fully evicted the BTB before the spy executes.

Covert channels on clean systems with only the spy and trojan executing are cool, but not necessarily practical in a real attack. In the real world a computer has all sorts of programs running which may use the BTB and create noise in the measurements of the spy. For my experiments I used a Firefox browser playing a youtube video to create this noise. Baseline noise levels (in red) were recorded without the trojan running.

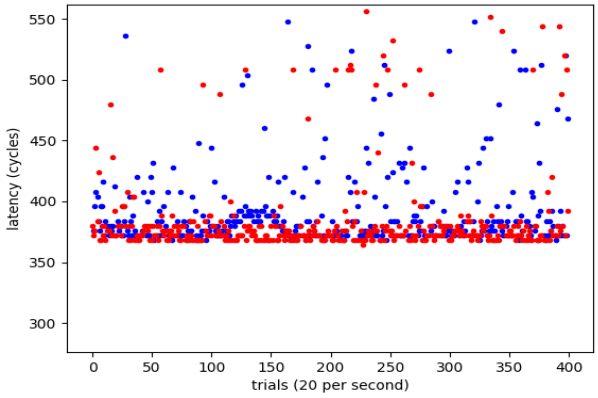

Figure 5. Spy and Trojan operating on the same virtual core with noise on the same virtual core. Noise in red.

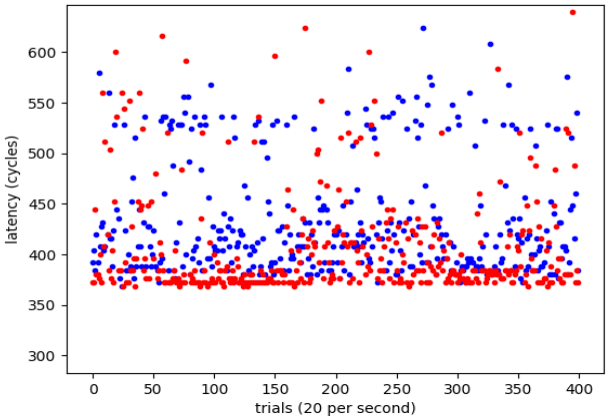

Figure 6. Spy and Trojan operating on the same virtual core with noise on the other virtual core. Noise in red.

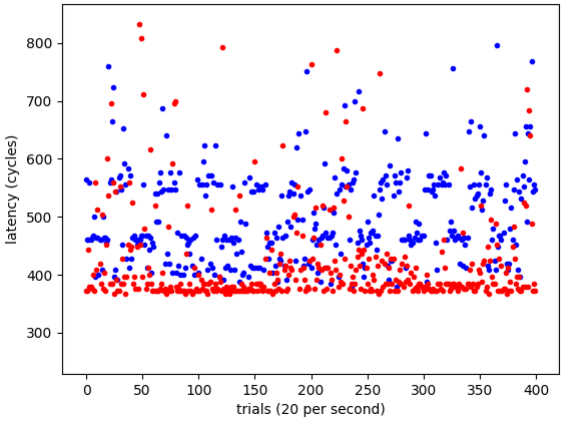

Figure 7. Spy operating on one virtual core and trojan and noise operating on the other. Noise in red.

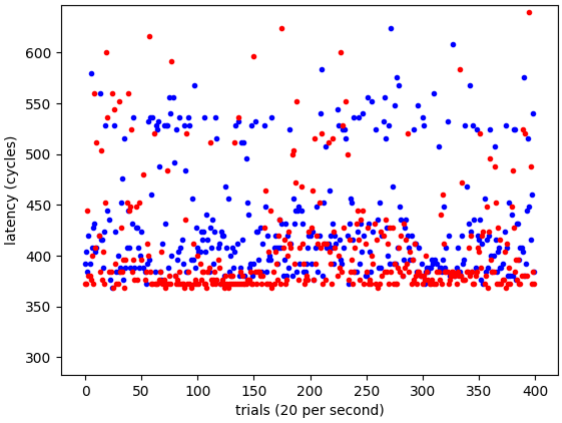

Figure 8. Trojan operating on one virtual core and spy and noise operating on the other. Noise in red.

The only channel that appears to remain readable is with the spy operating on one virtual core and the trojan and noise operating on the other. Optimizing the trojan and spy parameters may improve these results however.

4. MITIGATIONS

In “Understanding and Mitigating Covert Channels Through Branch Predictors” [4] they show that modifying the OS so it executes random branches during context switches is an effective way of resetting the branch prediction unit and preventing covert channels across it. The same method could be used here to flush the BTB. Their implementation also had little overhead, <20%, which could be reduced further by only flushing the BPU when the context switch is across security domains.

Another easy method proposed by many to prevent covert channels is simply not allowing programs of different security domains to execute via SMT on the same virtual core. Or even better just disable SMT altogether. This doesn’t eliminate covert channel across context switches however.

ACKNOWLEDGMENTS

Professor Evtyushkin for an excellent class in which I learned to never trust a computer.

REFERENCES

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

[2] ——. Predicting secret keys via branch prediction. In The cryptographers’ track at the RSA conference (2007).

[3] 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).

[4] D. Evtyushkin, D. Ponomarev and N. Abu-Ghazaleh, “Understanding and mitigating covert channels through branch predictors,” ACM Transactions on Architecture and Code Optimization (TACO), vol. 13, no. 1, p. 10, 2016.

[5] ——, “Covert channels through branch predictors: a feasibility study,” in Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy (HASP). ACM, 2015, p. 5.

[6] ——, "Jump over ASLR: Attacking branch predictors to bypass ASLR," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, 2016, pp. 1-13. doi: 10.1109/MICRO.2016.7783743

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

[8] M. Milenkovic, A. Milenkovic, and J. Kulick. Microbenchmarks for Determining Branch Predictor Organization. Software Practice & Experience, volume 34, issue 5, pages 465-487, April 2004.