# Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR

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

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

Nael Abu-Ghazaleh
Computer Science and
Engineering Department
University of California, Riverside
naelag@ucr.edu

***Abstract*—**

**Address Space Layout Randomization (ASLR) is a widely-used technique that protects systems against a range of attacks. ASLR works by randomizing the offset of key program segments in virtual memory, making it difficult for an attacker to derive the addresses of specific code objects and consequently redirect the control flow to this code. In this paper, we develop an attack to derive kernel and user-level ASLR offset using a side-channel attack on the branch target buffer (BTB). Our attack exploits the observation that an adversary can create BTB collisions between the branch instructions of the attacker process and either the user-level victim process or on the kernel executing on its behalf. These collisions, in turn, can impact the timing of the attacker's code, allowing the attacker to identify the locations of known branch instructions in the address space of the victim process or the kernel. We demonstrate that our attack can reliably recover kernel ASLR in about 60 milliseconds when performed on a real Haswell processor running a recent version of Linux. Finally, we describe several possible protection mechanisms, both in software and in hardware.**

***Index Terms*—Address Space Layout Randomization, Bypass, Side Channel, Timing Channel, Timing Attacks, Kernel Vulnerabilities, Exploit Mitigation.**

## I. Introduction

Memory corruption attacks such as stack and heap overflows [1], [2] and format string attacks [3] can lead to control hijacking and arbitrary code execution by the attackers. Despite significant efforts to prevent such attacks [4], [5], [6], [7], [8], they remain a serious exploitable class of vulnerabilities present in many types of software. Since creating a bug-free environment is practically impossible, systems are often hardened using techniques that substantially reduce the probability of a successful attack.

One such hardening technique is Address Space Layout Randomization (ASLR). ASLR provides protection by randomizing positions of key program components in virtual memory. The randomization targets code and data segments, stack, heap and libraries. The purpose of ASLR is to make it difficult, if not impossible, for the attacker to know the location of specific code pages in the program's address space. For example, even if the attacker successfully hijacks the control flow, it would be difficult to perform a meaningful return-oriented programming (ROP) [9], [10], [11] attack under ASLR, because the addresses of ROP gadgets to inject on the stack are not known due to randomization. Relying on brute-force solutions to discover required gadget addresses can cause the program to crash, or it can take prohibitively long time [12], enabling detection by system software [13]. Discovering and exploiting other vulnerabilities that disclose the randomization algorithm significantly complicates the attack [14]. Non-control-data attacks [15] require the attacker to know locations of various data structures. Although our attack directly recovers ASLR for the code segment only, data segments are typically not decoupled from code segments [16]; thus a successful attack on code ASLR reveals the locations of data structures. Today, ASLR-based defenses are widely adopted in all major Operating Systems (OS), including Linux [17], Windows [18] and OS X [19]. Smartphone system software such as iOS [20] and Android [13] also use ASLR.

ASLR implementations across different operating systems differ by the amount of entropy used and by the frequency at which memory addresses are randomized. These characteristics directly determine the resilience of ASLR implementations to possible attacks. For example, 32-bit systems have a much smaller addressable space, limiting the amount of space that can be dedicated to randomization, making it possible to build fast brute-force attacks [12]. The randomization frequency can range from a single randomization at boot or compile time to dynamic randomization during program execution. More frequent re-randomization reduces the probability of a successful attack.

Traditionally, ASLR has only been considered as a protection mechanism against remote attacks. As a result, and also for performance reasons [21], some ASLR implementations randomize positions of libraries only one time during the system boot. Consequently, all processes executed on a machine receive the same mappings of the libraries, thus making

978-1-5090-3508-3/16/$31.00 

return-to-libc [22], [23] or other code reuse attacks possible within the same system. The presence of many high-privileged processes in the system makes the attack surface large. For example, on our experimental machine, an OS with only basic services executes 30 and 80 root background processes in Ubuntu 14.04 LTS and OS X El Capitan 10.11.2 respectively. If one of these processes is subverted by an attacker, the entire system becomes compromised with respect to ASLR.

All current operating systems supporting randomizations implement variants of ASLR for both user and kernel-level address spaces. Kernel-level ASLR (KASLR) randomizes kernel code segments and can stop attacks that require knowledge of the kernel address space layout (including ROP, jump-oriented programming (JOP), return-to-libc, ret-2-user [24] and other attacks). Unfortunately, current implementations of KASLR are often criticized for incompleteness and insufficient entropy [25]. A small entropy is typically justified by the fact that it is infeasible for an adversary to mount a brute-force attack against KASLR. If the attacker guesses the randomization incorrectly, the kernel typically crashes and the attack fails.

In this paper, we demonstrate a new attack that can recover all random bits of the kernel addresses and reduce the entropy of user-level randomization by using *side-channel information* from the Branch Target Buffer (BTB). Our attack only requires the control of a user-level process and does not rely on any explicit memory disclosures. The key insight that makes the new BTB-based side-channel possible is that the BTB collisions between two user-level processes, and between a user process and the kernel, can be created by the attacker in a controlled and robust manner. The collisions can be easily detected by the attacker because they impact the timing of the attacker-controlled code. Identifying the BTB collisions allows the attacker to determine the exact locations of known branch instructions in the code segment of the kernel or of the victim process, thus disclosing the ASLR offset.

Our attacks exploit two types of collisions in the BTB. The first collision type, exploited to bypass KASLR, is between a user-level branch and a kernel-level branch - we call it *cross-domain collisions*, or CDC. CDC occurs because these two branches, located at different virtual addresses, can map to the same entry in the BTB with the same target address. The reason is that the BTB addressing schemes in recent processors ignore the upper-order bits of the address, thus trading off some performance for lower design complexity. The second type of BTB collisions is between two user-level branches that belong to two different applications. We call these collisions *same-domain collisions*, or SDC. SDCs are used to attack user-level ASLR, allowing one process to identify the ASLR offset used in another. An SDC occurs when two branches, one in each process, have the same virtual address and the same target.

We demonstrate our attack on a real system with Haswell CPU and a recent version of Linux kernel equipped with ASLR. Since this new attack adds to the arsenal of a potential adversary, we also discuss a number of possible software and hardware-supported mitigation mechanisms to thwart this attack. The solutions range from further hardening the ASLR implementations to reconsidering the hardware designs of the BTB to avoid collisions.

In summary, this paper makes the following contributions:

- We describe a new technique to bypass existing ASLR schemes by exploiting a side-channel created through shared BTB. We show how an adversary can create a robust side-channel between a user process and the kernel, as well as between two user processes in a controlled manner.
- We show how the details of the BTB addressing scheme, needed for creating a reliable BTB side channel, can be reverse-engineered.
- We show how the new BTB side-channel attack can be used to recover kernel and user-level ASLR in a fast and reliable fashion. We implement our attacks on a real system with Haswell CPU and recent Linux kernel and show that kernel-level ASLR can be recovered in about 60 milliseconds.
- We propose several software and hardware countermeasures against the new attack and also place our attack in the context of the related work.

## II. Threat Model and Assumptions

We consider two distinct attacks/threat models: one on KASLR and one on user-level ASLR. In this section we describe our assumptions about the underlying system and the capabilities of the attacker in each case.

### A. KASLR Attack

We assume that an attacker has control over a process running on the target system with normal user privileges. We refer to this process as the spy process. We further assume that the kernel implements some form of KASLR and that the kernel's code and module segments are randomly placed during boot in accordance with the KASLR algorithm. The attacker knows which bits of the address are randomized and which bits are fixed. Beyond that, the attacker does not have any knowledge on how the random bits were generated. The spy process can only execute normal user-level instructions, including instructions for time measurement (such as `rdtsc`) and perform regular interactions with the kernel through system calls. We assume that the spy process cannot brute-force the correct address. We do not assume any weaknesses in KASLR implementation: in other words, randomized offset values were generated using strong sources of entropy with no algorithmic weaknesses [26]. The goal of the attacker is to recover the address of a kernel routine, such as a system call handler in virtual memory.

### B. Attack on User Process ASLR

In this case, we assume that two user processes are present in the system. The first process is a process that is the target of an attack typically because it runs as root or has permissions to access sensitive data. We refer to this process as the victim process. The other process is the spy process and it is controlled by the attacker who seeks to recover ASLR of the victim as a first step of further attacks on it. We assume that the system supports ASLR and the victim is compiled as a position-independent executable to fully benefit from it. As a result, the segments in the process image are randomly shifted in virtual memory. We assume that there are no weaknesses in the ASLR implementation and the spy does not have any means to directly obtain the randomized ASLR offset, for example via a memory disclosure vulnerability. The spy can only execute normal user-level instructions, including instructions for time measurement. The spy can also perform regular interactions with the victim via any interfaces it offers, for example via initializing network connections.

We assume that the spy can achieve virtual core co-residency with the victim process. This assumption is required because BTB collisions can only be achieved within the same virtual core on Haswell CPU. This assumption is not unrealistic because, as we demonstrate in Section V-A, it is possible for the spy to control the cores on which the victim process is scheduled.

## III. Creating the BTB Side-Channel

Branch predictors are critical to performance of modern processors. One of the main components of the branch prediction hardware is the Branch Target Buffer (BTB). The BTB stores target addresses of recently executed branch instructions, so that those addresses can be obtained directly from a BTB lookup to fetch instructions starting at the target in the next cycle. Since the BTB is shared by several applications executing on the same core, information leakage from one application to another through the BTB side-channel is possible. For example, several previous works demonstrated the feasibility of recovering secret encryption key bits using branch predictor side channel [27], [28]. As another example, a recent study of [29], [30] demonstrated that a reliable and high speed covert communication channel can be created between two malicious applications that share the branch prediction logic.

In this paper we describe a new security threat associated with shared branch prediction hardware. Specifically, we demonstrate how a user-level spy process can gain information about the position of code blocks in the address space of either a victim process or the kernel by aligning its code to intentionally create BTB collisions between these two address spaces. The attacker performs a series of time measurements, each to test a hypothesis about the location of a specific branch. These experiments allow the attacking process to discover the precise location of a branch in the kernel address space, or the address space of another process, thus bypassing kernel-level ASLR and user level ASLR respectively. In principle, our approach has a potential to bypass even some recently proposed fine-grained ASLR solutions [31], [32], [33], [34].

The BTB side-channel that we exploit in this paper for attacking ASLR is based on creating BTB collisions between unconditional branch instructions belonging to two different execution entities. We consider two types of collisions: collisions between two user-level processes (to attack user-level ASLR) and collisions between user-level and the kernel (to attack KASLR). While no specific BTB addressing details are needed by the attacker to perform the attack on the user-level process, some reverse-engineering and understanding of the BTB addressing scheme is required for an attack on the kernel.

### A. Creating BTB Collisions in User Space

To create a BTB-based side-channel, three conditions must be satisfied. First, one application has to fill a BTB entry by executing a branch instruction. Second, the execution time of another application running on the same core must be affected by the state of the BTB. This condition is satisfied when both applications use the same BTB entry, perhaps with different targets stored. Third, the second application must be able to detect the impact on its execution by performing time measurements. We call the BTB collisions created between two processes executing in the same protection domain (e.g. two user-level processes) as Same-Domain Collisions (SDC). An example of a SDC collision that can be exploited by our attack is shown in Figure 1.

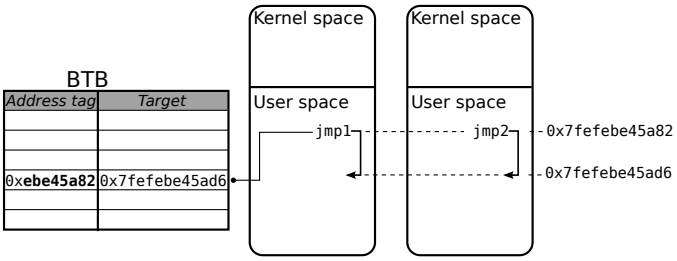



Fig. 1: SDC Example

To verify the sufficiency of the above assumptions for creating SDCs, we designed and conducted the following experiment on a machine with an Intel Haswell processor. We executed two processes on the system: the victim and the spy. The victim process writes some data into the BTB by executing branch instructions. The goal of the spy process is to use the branch instructions and time measurement tools to detect the information that was written to the BTB by the victim process. Since the goal of this experiment is to demonstrate the possibility of data transmission through the BTB side-channel, we allow the victim and the spy to coordinate their actions by communicating with each other using

signals. The BTB manipulations are performed by executing the `jmp` instruction. Conditional branches can also be used, but for the sake of simplicity we demonstrate our attack using unconditional jumps.

The key code block executed by both processes contains a single `jmp` instruction with additional `nop` instructions. The outline of this code is depicted in Listing 1. The code has two possible jump targets. However, when compiled, the target is statically set in the binary. We placed this code in both processes. The functions where this code was placed are called `spy_func()` and `victim_func()` accordingly.

```
asm("jmp target2;");
asm("nop; nop; ... ");
asm("Target1:");
asm("nop; nop; ... ");
asm("Target2:");
asm("nop; nop; ... ");
```

Listing 1: Code example with jump instructions

The goal of this experiment is to record the execution time of the spy process when the measured `jmp` instruction aligns with a similar instruction in the victim process. Our expectation is that the two branch instructions will map to the same BTB entry if they are located at the *same virtual address*. Assuming $BTB\_index()$ is the BTB indexing function, then $BTB\_index(A_{victim}) = BTB\_index(A_{spy})$ if $A_{victim}$ and $A_{spy}$ are identical virtual addresses. As a result, the data placed at this address by the victim will affect the execution time of the spy due to the created SDC.

The victim and the spy processes are compiled and linked specifically to align the jump instructions. The spy also measures the number of execution cycles required to execute the jump block using the `rdtscp` instruction. We adjust the number of `nop` instructions in the spy in order to compensate for the length of the `rdtscp` instruction and keep the addresses of the jump and both targets aligned with the victim's addresses. The disassembly of the key functions is depicted in Listing 2.

We executed the spy and the victim processes under two settings. In the first setting, the targets of the spy and the victim are the same, as demonstrated in Listing 2. In the second setting, the targets are different and the victim jumps to target **T1**. The use of different targets results in extra BTB misses and thus performance slowdown for the spy, as the spy's BTB entry is overwritten by the victim, but with a different target address. By running the spy under both settings, we obtain the number of cycles required by the spy to execute the jump code block.

The timing diagram depicting the stages of the experiment is presented in Figure 3. The affinity masks of the two processes are set to force them to execute on a single virtual core interchangeably. First, the spy process sends a signal ① to the

```
3000 <victim_func>:
3000 push %rbp
3001 mov %rsp,%rbp
3004 nop
3005 nop
 .   .   .
3021 jmp <T2>
3023 nop
 .   .   .
302d <T1>
302d nop
 .   .   .
3037 <T2>
3037 nop
 .   .   .
3041 pop %rbp
3042 retq
```

```
3000 <spy_func>:
3000 push %rbp
3001 mov %rsp,%rbp
3004 rdtscp
3005 nop
 .   .   .
3021 jmp <T2>
3023 nop
 .   .   .
302d <T1>
302d nop
 .   .   .
3037 <T2>
3037 nop
 .   .   .
3041 rdtscp
 .   .   .
306c pop %rbp
306d retq
```

Listing 2: Disassembly of the functions containing the jump block in the victim and the spy. Pictured an example with aligned jump and target addresses.

victim process. To assure the delivery and the correct response before the spy continues, the spy process calls the `sleep()` ③ function immediately after sending the signal. The victim handles the signal by executing the `victim_func()` ② function. At this stage, an entry in the BTB will be created or updated. After a short sleep, the spy process executes the `spy_func()` ④ and measures the number of cycles to execute its jump block. After this step, the measurement process is repeated again. When enough measurements are obtained, the same experiment is repeated with the victim jumping to a different target. We ran this experiment to generate 100,000 measurements under each setting.

| | **Spy T2** | **Spy T1** |
|---|---|---|
| **Victim T2** | 55.76 | 69.38 (+11.12) |
| **Victim T1** | 64.93 (+9.17) | 58.26 |

TABLE I: Averaged time of the jump code block (in cycles) as measured by the spy with different victim settings.

The results of this experiment are shown as a histogram in Figure 2. As can be observed from the graph, there are two separate groups of time measurement values. The difference between the averages of these two groups is about 9 cycles. According to Intel developer's manual, the frontend resteer following an incorrect BTB prediction introduces an 8-cycle bubble into the instruction fetch pipeline. This value is similar to the observed slowdown.

In order to verify the consistency of our results, we also repeated the experiment with the spy having the jump target

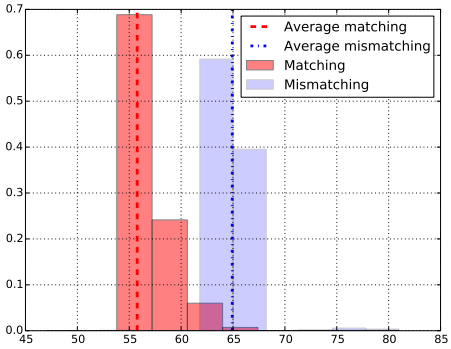



Fig. 2: Distribution of the execution time of the jump code block (in cycles) when victim process jumps to matching and mismatching target addresses.

set to **T1**. The results are summarized in Table I. Note that since **T1** is located closer to the jump instruction in the code segment, the spy executes more instructions when jumping to **T1**. Consequently, **T1**'s measurements are higher comparing to **T2**'s with the victim performing jumps to the matching target. However, the relative difference between the matching and mismatching targets is similar in both cases.

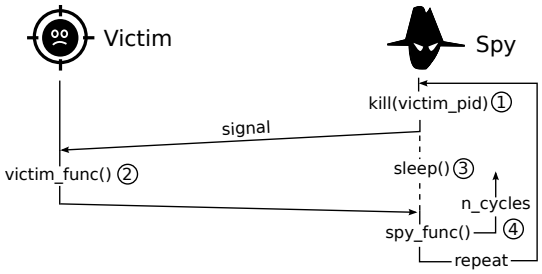



Fig. 3: Interactions between the spy and the victim

These results validate our hypotheses that the BTB access performed by one process impact the execution time of another process. The stability of the results offers an opportunity to use the BTB as a side-channel to monitor the branch activity of a victim process.

### *B. Creating Cross-Domain BTB Collisions*

We now describe how the BTB collisions can be created between a user-level process (the attacker) and the kernel as it executes on behalf of the user process. We call such collisions Cross-Domain Collisions (CDC) because the spy and the victim belong to different protection domains. If the full virtual address was used for BTB addressing, then CDCs would not exist, because the kernel code and the user code are located at different virtual addresses. In this case, the BTB tags would be different and the user process would never experience a BTB hit on a data placed in the BTB by the kernel.

However, modern processors typically use long (48-bits in current implementations) virtual addresses when they run in the 64-bit mode [35]. Assuming that each BTB entry also needs to store the absolute value of the target address, the total size of each BTB entry becomes large if the entire virtual address is used for tagging and indexing. For example, for a BTB with 8K sets, 35 bits for the tag and 48 bits for the target address will be needed, adding up to 83 bits of storage for each entry, which is expensive and power-consuming. Therefore, current CPUs typically store only a part of the upper-order address bits as tags, and some upper-order bits are ignored for BTB addressing, possibly creating more collisions but significantly simplifying the design and the BTB area and power requirements.

The knowledge of the precise addressing scheme in the BTB has significant implications on both types of attacks that we consider in this paper. For the user-level attack that exploits SDCs, this information is important because the address bits that are not used in BTB addressing cannot be recovered using our attack. In other words, this knowledge gives the attacker information about the maximum possible benefits of a successful attack. In terms of attacks against KASLR that exploits CDC, the knowledge of the addressing mechanism is essential even for creating the CDC collisions themselves.

On the Haswell processor used in our experiments, only a subset of the virtual address bits is used for BTB addressing. We used the following algorithm to discover which particular bits are used for addressing the BTB. First, we created a detectable SDC between two jump instructions in two separate processes (as described in the previous section). Second, by changing the address bits in the colliding instructions, we determined if the specific bits were used for BTB addressing. In particular, when inverting a specific address bit eliminates a previously observed collision, the conclusion is that this bit was used in the BTB addressing; otherwise it was not used.

To determine the relevant bits for Haswell processor, we repeated the experiment similar to the one described above in Section III-A under different settings. The main difference was that while we kept the address of the jump instruction the same in the victim process, we changed the address bits of the jump instruction in the spy process. We discovered that all lower bits of the address are used continuously and only the higher bits of the address were cut off and not used in the BTB addressing. In particular, bits 0 to 30 of the virtual address are used, and bits 31 to 47 are ignored. The reverse engineered BTB addressing scheme in the Haswell processor is depicted in Figure 4. Figure 5 shows how a kernel-level branch instruction and a user-level branch instruction can create a CDC in the BTB.

Equipped with the understanding of the BTB addressing scheme, the attacker can now create CDCs in the BTB and exploit them for the kernel-level attack. The kernel-based

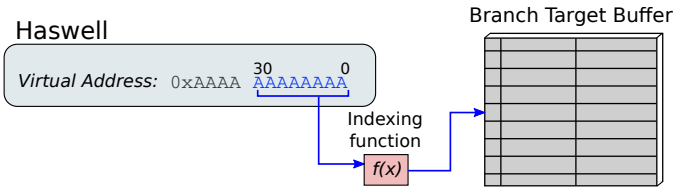



Fig. 4: BTB addressing scheme in Haswell processor

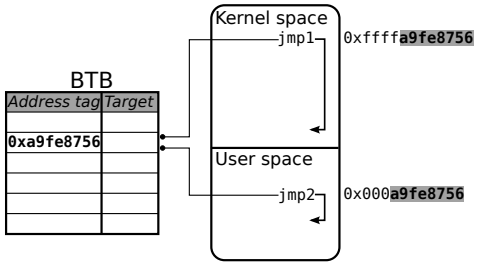



Fig. 5: CDC Example

attack flow is described in detail in the next section. After presenting the kernel-level attack, we show how a complete attack on user-level process can be realized using SDCs in the BTB.

## IV. Recovering ASLR Bits of Kernel Code Addresses

In this section, we demonstrate how the BTB side-channel can be used to effectively recover the randomized offset bits used in kernel ASLR in a short amount of time. We use a recent Linux kernel (version 4.5) for our experiments.

### A. KASLR in Linux

Modern desktop and server operating systems implement kernel ASLR (KASLR). Starting from kernel version 3.14, KASLR is also included in the mainstream Linux kernel distribution. To provide the necessary background, we first explain the KASLR implementation in 64-bit mode Linux kernel.

When KASLR is disabled, the kernel image is always placed at the same physical address during system's boot. The address translation in kernel mode can be performed by simply subtracting a predefined `PAGE_OFFSET` value (which is `0xffffffff80000000` for a 64-bit kernel) from the virtual address. Thus, the location of the kernel image is fixed in both the virtual and physical address spaces.

When KASLR is enabled, a sequence of random bits is generated during early boot process. These bits are used to calculate the randomized offset at which the kernel image is placed in physical memory. The virtual-to-physical address translation for kernel addresses remains unchanged. The randomized placement of kernel code in physical memory is mirrored by the same offset being applied to the virtual addresses in virtual memory. This leads to a critical observation: if an attacker discovers the position of the kernel code either in physical or virtual address space, the address layout is disclosed, and the location of any address in the static kernel image can be determined from there. Since our attack is built around virtually-addressed BTB, for the rest of this section we focus on virtual addresses. Note that since the kernel is mapped into the address space of every process, deriving the KASLR offset on any process exposes KASLR for all.

Due to the specifics of the Linux kernel memory layout, the 64-bit kernel currently randomizes only 9 bits of the virtual addresses. In particular, the kernel code must be aligned at 2MB boundaries. As a result, only the Page Directory Entry (PDE) bits of virtual addresses are randomized. While it would have been possible to extend the relocation mechanism, this requires significant reorganization of the physical memory layout and will likely result in performance degradation. Figure 6 demonstrates the randomized bits as well as how the bits are used for page translation on x86_64 machines operating with large 2MB pages. The figure also shows possible range of kernel code addresses.

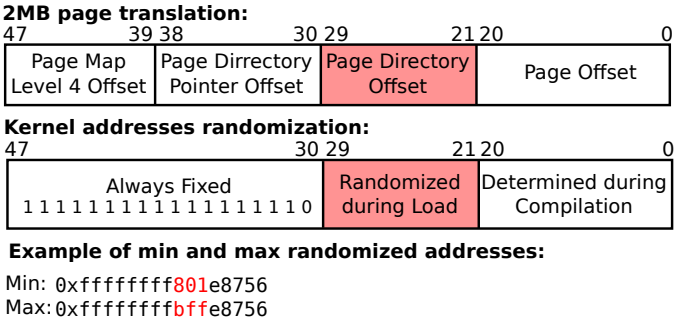



Fig. 6: KASLR used in 64-bit Linux kernel. Only the Page Directory Entry (PDE) bits are randomized.

### B. Using the BTB Channel for KASLR Bits Recovery

In Section III we demonstrated how two independent jump instructions can collide inside the BTB to create contention for BTB entries, resulting in a measurable slowdown of the colliding jump instructions. We now describe how this side-channel can be used to discover the random address bits of one kernel function. This, in turn, allows the discovery of a randomized offset value generated during the boot process.

To prepare the attack, an adversary needs to locate a branch instruction whose execution can be easily triggered by the spy process. One way to achieve this is to analyze the code of system calls available to a user process and locate a system call that performs a branch instruction. In order to make the attack faster and to minimize noise, first consideration should be given to system calls with a small number of instructions.

Next, the attacker creates a list of all possible locations for that branch taking into account the randomization scheme and the location of that branch in the compiled kernel code. After that, *for each address* **A** from that list, the attacker performs the following steps:

1) It allocates a buffer at the required address in the spy process.
2) It loads this buffer with a block of code containing a single jump instruction. The loading is done in a way that creates collisions in the BTB with a possible kernel branch instruction at address **A**. This block of code also contains an instruction to measure the time to execute the jump instruction.
3) The target branch instruction in the kernel code is activated by executing the identified system call.
4) The block of code in the spy process is executed a number of times and the number of cycles taken to execute it is recorded.
5) Finally, the results are analyzed. The block with higher average cycle measurement corresponds to the situation when the jump instruction in the spy's code block collides with the kernel branch at address **A**.

### C. Results of KASLR Bit Recovery

We implemented the attack discussed above and executed it on our test machine equipped with a Haswell CPU. In our experiments, we used the `open` system call to locate a branch instruction in the kernel code. This system call opens a file and returns a file descriptor. However, prior to accessing the file system, the OS checks for any errors. This check involves a comparison performed using a conditional branch instruction. The branch instruction is located at a predictable address and therefore can be used for the attack.

One of the possible errors that occurs during the system call is when the provided file name is larger than the maximum length allowed by the system. In order to make the attack fast and to minimize noise, we intentionally provided the erroneous filename. After the kernel detects the length violation, the control flow is immediately returned to the user process that requested this system call.

The results of this experiment are presented in Figure 7. The nine random address bits correspond to 512 possible **A** addresses. For each such address **A**, we collected 50 measurements. As seen from the graph, there is a single point that has the average timing that is much higher than the rest of the points on the graph. This point corresponds to the colliding branch instruction. In particular, the user space jump instruction at address `0xa9fe8756` collides with another branch instruction at address `0xffffa9fe8756`. This collision results in a significant slowdown due to the wrong-path instruction fetch after the BTB prediction. Therefore, the value of the KASLR bits is `a9f` in this case. The described attack takes a very short time: only 60 milliseconds are needed to collect the required number of samples.

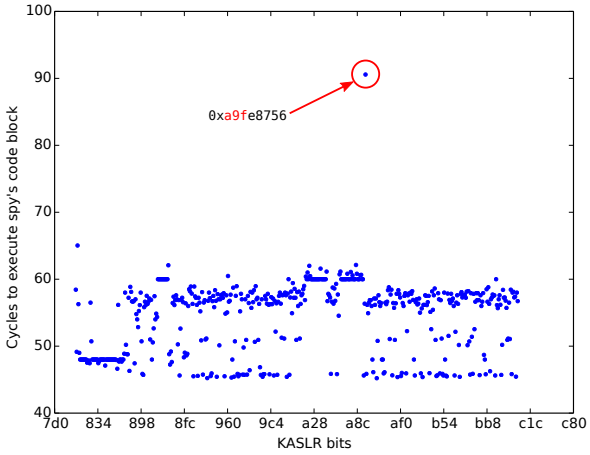



Fig. 7: Results of the BTB-based Attack on KASLR

## V. Recovering ASLR Bits in User Application Code

User-level processes can also be the victims of attacks attempting to compromise ASLR. Privileged processes or processes that have access to specific data can be attacked by other processes running in the same system; as part of such an attack, reverse engineering the ASLR offset is necessary. While system security settings may only allow an attacker to remotely create a process with user privileges, the attacker can use this user process as a starting point for discovering and attacking other processes with higher privileges.

A typical system has many daemon processes executing with administrative privileges and these daemons can have common exploitable vulnerabilities. For example, a recently reported vulnerability [36] in the `cupsd` printing scheduler (which is found on most Unix-like systems and is executing as root), allows a remote attacker to execute arbitrary code. Typically, ASLR interferes with the attacker's ability to perform a subsequent attack, such as a code reuse attack, that follows the exploitation of the vulnerability e.g. by means of buffer overflow. To complete a successful attack, the adversary first needs to de-randomize the victim process prior to enabling the correct exploitation of the vulnerability without crashing the victim process.

Another relevant application of a user-level ASLR attack is to perform layout de-randomization in isolated execution environments, where application secrets are protected inside enclaves or compartments [37], [38], [39], [40]. The compartment code layout is usually kept secret in such systems, and de-randomizing it opens avenue for side-channel attacks on compartments, similar to the ones reported by Xu et al. [41].

The BTB side-channel presents one way of de-randomizing the code segment of a running process in this scenario. Our contribution is to demonstrate the principles of such user-space attack. Similar to the attack on kernel ASLR described in previous section, the attack on user-level ASLR is based on the ability of the spy process to trigger some activity in the victim process. Such triggering can be accomplished in

several different ways depending on the interfaces offered by the victim process. For example, with the `cupsd` daemon, the spy can send some requests via the network to force some code to be executed as needed. In the attack preparation stage, the attacker analyzes the victim's executable to find functions that can be triggered and also locates jump instructions in such functions. The stages of the attack are presented in Figure 8. Specifically, the spy process performs the following steps for each possible address **A** where the victim's branch instruction can reside:

1) It allocates a buffer at the required address.
2) It fills the buffer with the code containing a single jump instruction at address **A**.
3) It triggers ① an activity in the victim process ② in order to force the victim to create a BTB entry.
4) It waits ③ for the activity to complete.
5) It executes the jump block several times and measures the execution time ④.
6) It changes the target of the jump instruction ⑤ and repeats the measurement stages.
7) Finally, the spy discovers the address at which the behavior of the jump is similar to what we described in Section IV.

We now discuss the requirements for this attack and mechanisms that allow the attacker to fulfill these requirements.

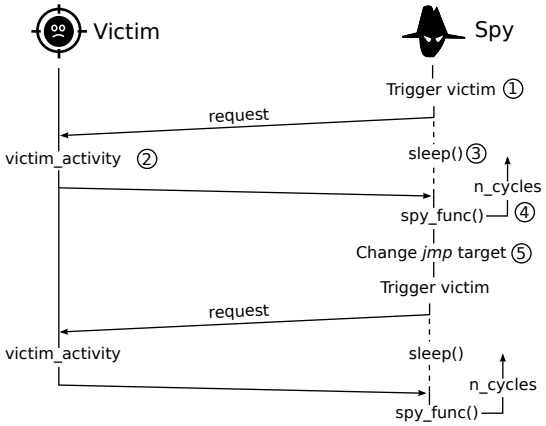
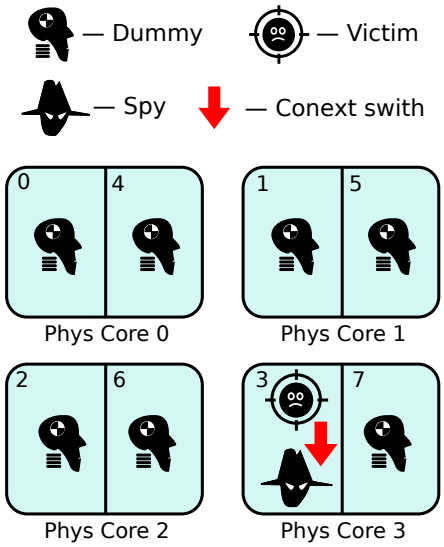



Fig. 8: Stages of the user ASLR attack

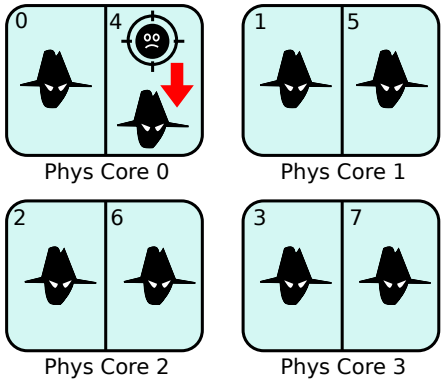
### A. Achieving Co-residency of the Victim and the Spy

The attack on kernel ASLR described in previous section does not have to be tied to a specific core. When a process performs a system call, the process switches to kernel mode and executes the kernel code which is mapped into its address space; i.e., the kernel code executes on the same core without causing a context switch. After that, the kernel returns to the process and allows it to continue.

The situation is quite different for the user-level attack. Our experiments with the BTB side-channel on both Sandy Bridge and Haswell CPUs revealed that a BTB collision between two user-level processes can only happen when both processes are




(a) Dummy processes force the OS to schedule the victim on a desired virtual core (prior the spy)




(b) A copy of the spy process is executed on each core

Fig. 9: Two types of spy scheduling. Both produce context switches from the victim to the spy allowing leakage of sensitive data.

scheduled consecutively on the same *virtual core* (or hardware thread context) of a hyper-threaded processor. The collision does not happen when the two processes are executed on parallel hardware contexts. This observation implies that either each virtual core has a dedicated area in the BTB, or every BTB entry is marked with a virtual core ID. Consequently, the first goal of the user-level attack is to ensure co-residency of the victim and the spy on the same virtual core.

The virtual core co-residency requirement can be met in several ways. One method is to inject dummy processes in the system on all other virtual cores in order to alter the scheduling mechanism and force the spy to be placed on the same virtual core as the victim. Another option is to execute the spy on all virtual cores. These two schemes are summarized in Figure 9 and are described below.

*1) Manipulating the OS Scheduler:* One way to achieve co-residency is to manipulate process scheduling by the kernel. We assume that it is possible for a user process to schedule itself on an arbitrary core. In Linux, such capability is available through a call to `sched_setaffinity()` [42]. Scheduling manipulation is possible because of predictability of the task placement algorithm. In order to efficiently and equally utilize the CPU computational and power resources, the OS attempts to spread the load equally among all cores. With that constraint, the OS is more likely to schedule a process on a core with a lower utilization level. To exploit this algorithm, the attacker can substantially load all cores in the system with dummy tasks, except for the core on which it wishes to place the victim process.

To validate that it is possible for a user-level process to control the placement of other processes (including privileged ones) in the system, we examined the scheduling of the `sshd` daemon process (OpenSSH server) that was executed as root. First, we did not introduce any dummy processes in the background and allowed the OS to freely choose any cores to schedule this process on. We performed a series of observations recording which core the process was scheduled on. During each observation, we initialized a new `ssh` connection in order to force the OS to wake up the process and place it on one of the cores. For the second part of the experiment, we executed dummy CPU-bound processes (as shown in Figure 9a), loading all but a single virtual core to the maximum. We repeated the same series of observations, recording the cores on which the `sshd` process was placed.

We executed the above experiment to capture 1 million data points for each case. For this experiment, we used a machine with Intel Core i7-4800MQ quad-core processor. Note that although the CPU has 4 physical cores, the hyper-threading technology makes the OS recognize 8 virtual cores with 2 threads per physical core. Our machine was running Ubuntu 14.04 LTS with the generic Linux kernel version 3.16.0-48.

The results of these experiments are presented in Figure 10. As seen from the graphs, when there are no dummy processes running on the CPU, the OS spreads the load among the cores equally. However, when there is a core with much lower utilization level, the OS would typically schedule the `sshd` daemon on that core. Another interesting observation is that the OS is more likely to choose virtual cores 0 – 3 (thus spreading the load among all four physical cores) before using virtual cores 4 – 7, which add another process on the same physical core via hyper-threading.

*2) Executing Multiple Spies:* An alternative method of achieving context switches from the victim to the spy on the same virtual core is to execute multiple copies of the spy process and allowing the OS to freely choose any core to schedule the victim process on. Figure 9b illustrates this approach. Even if a task migration happens, the victim process will be placed on a core with a copy of the spy process running. This method requires a slightly more complex spy organization. In order to achieve a continuous side-channel, the scheduled group of spy processes needs to detect which one of the spies is co-located with the victim process at the moment. They also need to communicate this information to each other. Detecting co-residency can be done either by directly obtaining such information from the OS or by relying on side-channel analysis [43].

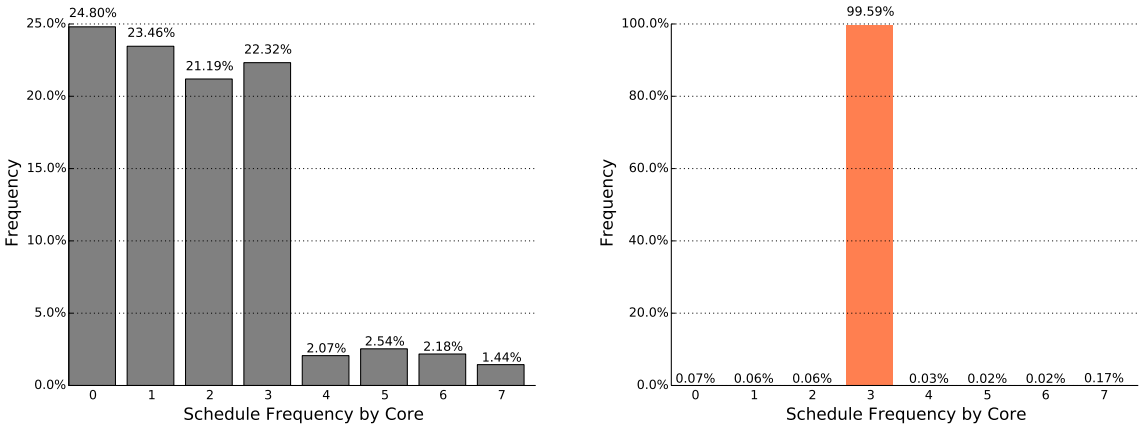
In our experimental system, which was configured with the default parameters, the OS allows all user processes to obtain the placement information on any other process, including privileged processes. This information includes the status of the process (active, sleeping, etc.) and the core number on which the target process is/was executing. Such information is provided through the `/proc` [44] virtual file system. This makes the detection of core co-residency straightforward. Moreover, the OS is more likely to reschedule a process to the same core due to scheduling policies favoring cache affinity [45], [46]. This scheduling approach reduces the number of times the process is migrated between the cores and promotes uninterrupted collection of side-channel data.

### *B. Experimental results*

We implemented a prototype of the attack on user-level ASLR by modifying the experiment described in Section IV. First, we compiled the victim process with full ASLR support. Second, we equipped the spy process with the capability to check possible locations of the victim branch. The victim has a jump with target **T2**, while the spy repeatedly tries two targets: the *matching* target **T2** and the mismatching target **T1**. The results are presented in Figure 11. For demonstration purposes, we only show the recovery of 8 bits of the address.

The results are obtained following the methodology of Section 3.1. The victim and the spy code is the same as shown in Listing 2. The spy executes more instructions when jumping to target T1, therefore the blue points are higher than the green points on Figure 11. The vertical line in the middle of the graph shows the situation when the victim's branch and the spy's branch hash to the same BTB entry, causing a collision and additional delay at the spy. While the latency increase in T1 is expected, the slight increase in T2 (matching target) is due to collisions and contention on other shared virtually-addressed resources, such as the uop cache. The collisions are not through BTB in this case because both victim and spy will get the BTB hits for the same virtual address and the same target address (T2).

Our prototype code tests 100 addresses in a second. Further optimizations can make the throughput even higher. Please note that current BTB addressing scheme (as used in Haswell processor used for our experiments) allows us to recover only a limited number of ASLR bits. The number of bits that are




(a) No scheduling manipulation

(b) The spy forces scheduling at core number 3

Fig. 10: Scheduling of the victim process to virtual cores.

randomizes is implementation specific. However, according to [47], the full ASLR in Linux randomizes 12th to 40th bits of the virtual address. Since 30th and higher bits are not used in BTB addressing, only 18 bits can be recovered using the BTB attack on Haswell. However, this significantly reduces the entropy, making the brute force of the remaining 11 bits more feasible.

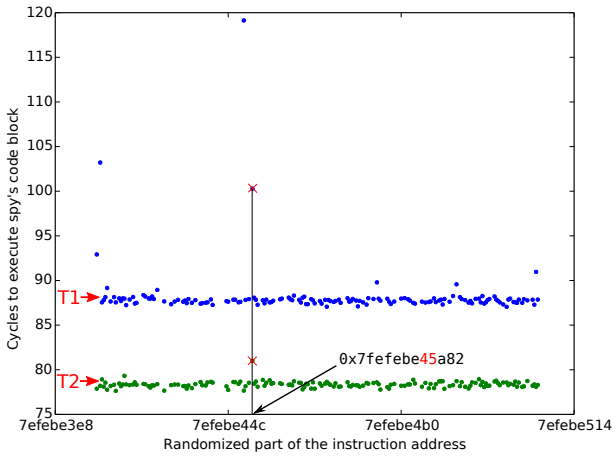



Fig. 11: Results of the BTB-based Attack on User-level ASLR

## VI. Mitigating BTB-based Attacks

In this section we describe several possible countermeasures that can either make the ASLR schemes less vulnerable to the BTB side-channel attacks presented above, or completely suppress the side-channel. We categorize possible solutions into two groups: purely software solutions and hardware-supported solutions.

### A. Software Mitigations

Software countermeasures are limited because they are not able to control how branches are mapped to the BTB entries, thus they do not address the root cause of the side channel. However, several software countermeasures are possible that can make the recovery of the ASLR bits difficult or even impossible.

Traditional ASLR schemes randomize only the offset of the segments within the program address space. However, recent research efforts suggested finer-grained ASLR schemes that can randomize code at the granularity of functions [31], basic blocks [48] or instructions [49], [50]. While most of these solutions focus on user-level ASLR, fine-grained KASLR is also possible. Reconstructing the code layout in memory is much harder when such fine-grained protection schemes are applied. The BTB attack described in this paper has a potential to bypass even these fine-grained techniques, provided that they preserve the basic block structure, because it discovers the position of individual branch instructions in memory. However, such an attack on fine-grain ASLR would require significantly more effort from the attacker. In addition, if an ASLR technique randomizes the sizes of basic blocks, the spy process will not be able to distinguish basic blocks from one another, thus closing the BTB side-channel.

Another approach to mitigating the BTB side channel (as well as many others) is to make the accurate execution time readings difficult by fuzzing the time stamp counter, or disabling it completely [51], [52]. However, such a solution could interfere with many legitimate applications that need to have precise time measurement capability. Moreover, some implementations have been shown to be insecure [53].

*1) Kernel ASLR Reinforcement:* As was demonstrated in Section IV-A using the example of Linux, today's operating systems rely on limited entropy for KASLR. A small number

of randomized bits makes the KASLR side-channel attack possible. In addition, the bits being randomized fall into the lower 30 bits of the virtual address[1], making the BTB-based attack possible. A simple solution which does not require any hardware changes is to ensure that more higher-order bits are randomized during every system load. The memory organization on x86_64 machines allows the use of 48 bits of virtual address with the most significant bit used to distinguish between the lower half and the upper half of virtual memory. Thus, assuming 2MB pages for kernel code (and thus 21 bits in the page offset), the maximum number of bits that can be theoretically randomized is 26 (48-1-21). Since 17 of these bits (the upper order bits of the address) are not used for BTB addressing, the BTB side-channel will not provide sufficient information to derive ASLR. The 17 bits correspond to approximately 131,000 possible kernel positions. Since brute-force attacks in kernel space are infeasible, this level of entropy is likely to provide sufficient security.

Randomization schemes interfere with the way the kernel normally organizes its memory layout. In current implementations, large memory areas are fixed and reserved for devices, the hypervisor and other service structures. Therefore, implementing such a scheme would require significant memory reorganization in order to benefit from high levels of entropy.

### B. Hardware Mitigations

The BTB side channel attack is possible because of two types of BTB collisions. The first type of collision occurs when two branches residing in different protection rings and located at different addresses are mapped to the same BTB entry - this opens the door for BTB-based attack against KASLR. The second type of BTB collision occurs when two different branch instructions are located at identical virtual addresses but belong to different processes - this collision is the basis for an attack on user-level ASLR.

A hardware solution that would fundamentally mitigate the BTB-based attacks is to change the BTB addressing mechanism in a way that prevents exploitable collisions in the BTB. The attack against KASLR can be mitigated by using full virtual address for accessing the BTB, thus eliminating collisions between the user code and the kernel code. This would require adding extra bits in the BTB, as the tag size will increase significantly (by 17 bits compared to Haswell implementation for 48-bit virtual addresses).

Alternatively, the BTB can use different indexing functions for user and kernel-level code. For example, a secret value can be added to the existing BTB hash function when the CPU is executing in the kernel mode. To prevent the user process from discovering this value and reverse-engineering the hash function, this value can be randomized during each system's boot.

[1]The number of bits can be different on other microarchitectures

In order to apply the same protection technique to mitigate user-level ASLR attack, each process needs to have a unique value that will be used in the BTB hashing function. This can be the hardware Address Space Identifier (ASID) [54], or the values of the `CR3` register, which are unique for each process. Of course, these values must be kept secret from other processes.

Other possible hardware-supported mitigation techniques include flushing the BTB on context switches or marking each BTB entry with unique process ID to distinguish the entries set up by different processes.

## VII. Related Work

Hardware side channels are well-known threats to security of sensitive data. A large number of prior works studied different aspects of side channels in instruction and data caches [55], [56], [57], [58], [59]. Aciicmez et al [27], [28]. were the first to demonstrate side channel attacks on branch prediction units to recover secret keys. In another recent work [30], [29], branch predictor's pattern history tables were used to build a covert channel and to pass information from one process to another. In this paper, we show how the specially-constructed new BTB side channel can be used to discover the memory layout of another process or the kernel, thus bypassing existing KASLR schemes and reducing the entropy of user-level ASLR.

Several works have demonstrated how ASLR can be defeated through brute-force approach [12], and by using memory disclosure attacks [60]. Some studies also showed how to bypass ASLR on mobile platforms [61] and novel shielded systems [41]. In [62], Gu et al. addressed the problem of de-randomizing kernel address space based on signatures generated from kernel memory snapshots.

Hund et al [63] demonstrated an attack on kernel ASLR using cache-based timing side channel analysis. In that work, three different attacks were demonstrated. The first attack relied on the collision of kernel and user objects in the last level caches. This attack requires the attacker to know the physical address of data that he places in the cache. In addition, the attack is hard to perform in realistic scenario because of excessive amount of noise in the last-level caches from other processor cores. The second attack described in [63] exploits a particular property of Intel's CPUs. In particular, when a user process tries to access a location in the kernel memory space, an exception is generated. Even though the access is denied, the TLB entry is still created, which can be detected later by the user process. This attack allows the spy process to discover which memory pages are allocated in kernel space. The third attack of [63] is based on observing the time needed for a page walk. To perform the cache-based attack, the spy process first flushes all cache levels and then invokes some system service. After that, it performs a memory access, times it, and uses this information to determine what kernel code

resides on what page. Compared to the cache-based attacks, the BTB-based attacks proposed in this paper are significantly simpler and allow the adversary to perform the attack under a much more controlled setting. Indeed, the attacker no longer needs to deal with noisy measurements from the last-level cache and does not require any knowledge about the physical addressing. All our attacks are based on virtual addressing and the exploitation of simple BTB collisions. While the attacks on KASLR using side-channel information were previously considered, as described above, to the best of our knowledge this paper is the first to consider side-channel analysis to recover user-level ASLR.

To harden the security properties of ASLR schemes against emerging attacks, researchers have proposed various fine-grain ASLR techniques. These schemes enforce the enhanced randomization at different levels, such as at the level of functions [31], basic blocks [48] or instructions [49], [50]. While some of fine-grain ASLR solutions can thwart our BTB attack by randomizing the relative positions of branch instructions in memory, they usually have significant performance overhead and are not widely adopted. In addition, our attack can still be used to locate the position of individual branch instructions in memory. This information can potentially lead to the development of other techniques to eventually reconstruct the code layout in memory despite deep randomization.

Snow et al. [64] presented just-in-time code reuse attack as an effective technique against sophisticated fine-grained randomization schemes. The attack is based by scanning and constructing the Return-Oriented-Programming (ROP) payload at the runtime exploiting memory disclosures. A system called Heisenbyte [65] has been proposed to protect against such just-in-time attacks by addressing memory disclosure attacks. Heisenbyte defeats memory disclosure attacks by introducing the concept of destructive code reads. Since our BTB attack does not rely on code reads from memory, it can still be used in an arsenal of tools to create memory disclosures and eventually reconstruct the address randomization schemes.

## VIII. Concluding Remarks

Address Space Layout Randomization (ASLR) is a widely-adopted security mechanism, both in kernel and application levels, to protect systems from code reuse attacks. In this paper, we exploited collisions in shared BTBs to create BTB side-channels and allow the attacker process to recover the memory layout of both the kernel and user-level applications. We demonstrated a successful attack on a system with Haswell CPU and a recent version of Linux kernel. We showed that our attack is robust and can bypass KASLR in a very short amount time. In addition, the attack can reduce the entropy of the user-level ASLR. Since this BTB attack adds to the arsenal of tools available to the attackers, we also described possible software and hardware countermeasures to mitigate this new security threat.

## IX. Acknowledgment

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

## References

[1] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[2] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129.

[3] T. Newsham, "Format string attacks," 2000.

[4] F. Qin, S. Lu, and Y. Zhou, "SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 291–302.

[5] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005, pp. 378–387.

[6] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic diagnosis and response to memory corruption vulnerabilities," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 223–234.

[7] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks." in *NDSS*, 2014.

[8] E. C. Sezer, P. Ning, C. Kil, and J. Xu, "Memsherlock: an automated debugger for unknown memory corruption vulnerabilities," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 562–572.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[10] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 2012, pp. 94–105.

[11] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from code reuse attacks," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 258–269.

[12] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[13] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev, "Address space randomization for mobile devices," in *Proceedings of the fourth ACM conference on Wireless network security*. ACM, 2011, pp. 127–138.

[14] "CVE-2015-3108." Available from NVD, CVE-ID CVE-2015-3108, Sep. 6 2015, [Online; accessed Feb. 2 2016 https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3108].

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *Usenix Security*, vol. 5, 2005.

[16] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 280–291.

[17] PaX Team, "Address space layout randomization," *Phrack, March*, 2003.

[18] M. Howard, "Address space layout randomization in Windows Vista," *Microsoft Corporation*, vol. 26, 2006.

[19] D. Keuper, "XNU: A security evaluation," December 2012.

[20] D. A. Dai Zovi, "Apple iOS 4 security evaluation," *Black Hat USA*, pp. 1–29, 2011.

[21] M. Payer, "Too much PIE is bad for performance," 2012.

[22] S. Designer, "return-to-libc attack," *Bugtraq, Aug*, 1997.

[23] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[24] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: lightweight kernel protection against return-to-user attacks," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 459–474.

[25] B. Spengler and PaX Team, "KASLR: An Exercise in Cargo Cult Security," Available online, Mar. 20 2013, [Online; accessed Feb. 2 2016 http://forums.grsecurity.net/viewtopic.php?f=7&t=3367#p12726/].

[26] T. Mandt, "Revisiting iOS Kernel (In) Security: Attacking the early_random() PRNG."

[27] 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.

[28] ——, "Predicting secret keys via branch prediction," in *The cryptographers' track at the RSA conference*, 2007.

[29] 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.

[30] ——, "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.

[31] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *null*. IEEE, 2006, pp. 339–348.

[32] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 475–490.

[33] H. Xu and S. J. Chapin, "Improving address space randomization with a dynamic offset randomization technique," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 384–391.

[34] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits." in *Usenix Security*, 2005.

[35] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron processor for multiprocessor servers," *IEEE Micro*, no. 2, pp. 66–76, 2003.

[36] "CVE-2015-1158." Available from NVD, CVE-ID CVE-2015-1158, Sep. 6 2015, [Online; accessed Feb. 2 2016 https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-1158].

[37] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

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

[39] ——, "Flexible hardware-managed isolated execution: Architecture, software support and applications," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2016.

[40] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[41] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," 2015.

[42] M. Kerrisk *et al.*, "SCHED_SETAFFINITY(2) linux programme's manual," Dec 2015.

[43] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 313–328.

[44] M. Kerrisk *et al.*, "proc(5) Linux Programmer's Manual," Dec 2015.

[45] V. Kazempour, A. Fedorova, and P. Alagheband, "Performance implications of cache affinity on multicore processors," in *Euro-Par 2008–Parallel Processing*. Springer, 2008, pp. 151–161.

[46] J. Torrellas, A. Tucker, and A. Gupta, "Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 24, no. 2, pp. 139–151, 1995.

[47] H. Marco-Gisbert and I. Ripoll, "On the Effectiveness of Full-ASLR on 64-bit Linux," 2014.

[48] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[49] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[50] S. Shamasunder, "On the Effectiveness of Heterogeneous-ISA Program State Relocation against Return-Oriented Programming," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, SAN DIEGO, 2015.

[51] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 118–129, 2012.

[52] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 490–505.

[53] S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, "Unraveling time-warp: What all the fuzz is about?" in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[54] B. Jacob and T. Mudge, "Virtual memory in contemporary microprocessors," *Micro, IEEE*, vol. 18, no. 4, pp. 60–75, 1998.

[55] D. Gruss, C. Maurice, and K. Wagner, "Flush+ Flush: A Stealthier Last-Level Cache Attack," *arXiv preprint arXiv:1511.04594*, 2015.

[56] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *null*. IEEE, 2006, pp. 473–482.

[57] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[58] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.

[59] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 72.

[60] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized libc," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 60–69.

[61] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened ASLR on Android," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 424–439.

[62] Y. Gu and Z. Lin, "Derandomizing kernel address space layout for memory introspection and forensics," in *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 62–72.

[63] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 191–205.

[64] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[65] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 256–267.