Poking Holes in Information Hiding

Angelos Oikonomopoulos
Vrije Universiteit Amsterdam
a.oikonomopoulos@vu.nl

Elias Athanasopoulos
Vrije Universiteit Amsterdam
i.a.athanasopoulos@vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Abstract

ASLR is no longer a strong defense in itself, but it still serves as a foundation for sophisticated defenses that use randomization for pseudo-isolation. Crucially, these defenses hide sensitive information (such as shadow stacks and safe regions) at a random position in a very large address space. Previous attacks on randomization-based information hiding rely on complicated side channels and/or probing of the mapped memory regions. Assuming no weaknesses exist in the implementation of hidden regions, the attacks typically lead to many crashes or other visible side-effects. For this reason, many researchers still consider the pseudo-isolation offered by ASLR sufficiently strong in practice.

We introduce powerful new primitives to show that this faith in ASLR-based information hiding is misplaced, and that attackers can break ASLR and find hidden regions on 32 bit and 64 bit Linux systems quickly with very few malicious inputs. Rather than building on memory accesses that probe the allocated memory areas, we determine the sizes of the *unallocated holes* in the address space by repeatedly allocating large chunks of memory. Given the sizes, an attacker can infer the location of the hidden region with few or no side-effects. We show that allocation oracles are pervasive and evaluate our primitives on real-world server applications.

1 Introduction

While Address Space Layout Randomization (ASLR) by itself no longer ranks as a strong defense against advanced attacks due to the abundance of memory disclosure bugs [1], it is still an essential foundation for more sophisticated defenses that use randomization to provide fast pseudo-isolation. Specifically, these defenses hide important sensitive information (such as shadow stacks [2], safe regions [3], or redirection tables [4]) at a random position in a very large address space. The underlying and crucial assumption is that an attacker is not able to detect the location of the hidden regions.

Thus, the strength of all these defenses hinges entirely on the ASLR-provided obscurity of the hidden region. Our research question is whether such trust in the randomization schemes of modern systems like Linux is justified. In particular, we show that it is not, and introduce powerful new primitives, *allocation oracles*, that allow attackers to stealthily break ASLR on Linux and quickly find hidden regions on both 32-bit and 64-bit systems.

Randomization for information hiding Most operating systems today employ coarse-grained ASLR [5] which maps the different parts of the process (the stack, heap, and `mmap` region) in random locations in memory. The amount of randomness determines the strength of the defense. As an extreme example, the entropy for the `mmap` base on 32-bit Linux is as low as 8 bits, which means that the region can start at 256 possible locations in memory. This is well within range of a relatively stealthy brute-force attack. On 64-bit machines, however, the entropy of the `mmap` region on Linux is 28 bits and brute forcing is no longer considered practical. Unfortunately, whatever the granularity and entropy, address space randomization is vulnerable to information disclosure attacks. For example, in the absence of additional defenses and given a single code pointer, attackers can easily find other code pointers and eventually enough code to stitch together a code reuse attack [1].

However, powerful new defenses have evolved that still rely on randomization, but this time for the purpose of hiding a secret region of memory in a large address space [2, 3, 4]. Typically, they ensure the confidentiality and integrity of code pointers (such as return addresses, function pointers, and VTable pointers) [3, 6]. As manipulating a code pointer is vital for an attacker to take control of the program, preventing unauthorized access to code pointers also prevents such attacks. Thus, instead of storing code pointers in the program code, the heap, or

the stack, they place them in an *isolated* memory region. For instance, some defenses store the return addresses on an isolated "shadow" stack. Such defenses work as long as attackers cannot access the isolated region.

While it is possible to isolate these regions using techniques such as Software Fault Isolation (SFI) [7, 8], most existing solutions adopt cheaper ASLR-based *pseudo-isolation*—presumably for performance reasons or since commodity hardware-supported fault isolation can dramatically limit the size of the address space [9]. In other words, they resort to *information hiding* by placing the region at a random location in a very large virtual (and mostly inaccessible) address space and making sure that no pointers to it exist in regular memory.

The role of ASLR in information hiding is quite different to its use in countering code-reuse attacks directly, since even a strong *read* or *write* primitive ceases to be trivially sufficient for breaking the defense. Specifically, hiding all sensitive pointers forces attackers to probe the address space repeatedly (with the number of probes proportional to the size of the address space) and risk detection from crashes [10], or other observable events [11]. While Evans et al. [12] show that problematic implementations relying on huge hidden regions are still vulnerable to crashless probing attacks, more advanced defenses are not [6]. Indeed, the many new defenses that rely on information hiding show that ASLR is widely considered to offer strong isolation.

Allocation oracles Unlike previous approaches, our attack does not revolve around probing valid areas of allocated memory. Instead, we introduce new primitives to gauge the size of the *holes* in the address space. The key idea is that once an attacker knows the sizes of the holes, she can infer the start of the hidden regions. In other words, even if all the pointers into the hidden regions have been removed, the sizes of the unallocated parts of the address space "*point*" into the hidden regions.

To gauge the sizes of the holes, we introduce *allocation oracles*: information disclosure primitives that allow an attacker to allocate large chunks of memory repeatedly and thus probe for the possible sizes of the largest hole in the address space. In most cases, she can use binary search to find the exact size after a handful of probing attempts. The pre-conditions for allocation primitives are the ability to make repeated, arbitrarily large memory allocations, and to detect the success or failure of such allocation attempts. For instance, the simplest oracle might be the length field in a protocol header that controls the amount of memory a server allocates for a request [13]. More reliably, the attacker may corrupt a value in memory that is later used as an allocation size. Assuming the attacker can distinguish between success and failure of the allocations, this primitive operates as an allocation oracle. We will show that such cases are common in real-world server programs.

Allocation oracles come in two main forms. *Ephemeral* allocation oracles perform allocations that have a short lifetime. For instance, a server which allocates memory for a client request and frees it after sending the reply. Ephemeral allocation oracles are the most effective in detecting the hidden regions. In the absence of ephemeral allocation oracles, we may find *persistent* allocation oracles. In this case, the allocation is permanent. This property alone makes attacks harder, but not impossible. In this paper, we present exploitation techniques and examples using either kind of oracle, as well as a powerful combination of the two. This combination allows an attacker to disclose the location of small hidden regions *arbitrarily located* in an *arbitrarily large* address space with *no crashes* or other detection-prone side effects.

Contributions We make the following contributions:

- We introduce new types of disclosure primitives, termed *allocation oracles*. Unlike existing primitives, allocation oracles do not work by accessing memory addresses, but instead probe the address space for "holes". We describe primitives for both ephemeral and persistent allocations, and show how to combine them to break information hiding.
- We describe a methodology to assist an attacker in easily discovering both ephemeral and persistent allocation primitives in real programs. We show that such primitives are very common in practice. When real-world instances of our primitives are imperfect, we show how an attacker can exploit *timing side channels* to mount effective attacks.
- We show that our primitives can be exploited to mount end-to-end disclosure attacks on several real-world server programs. Our attacks render ASLR ineffective even on 64-bit (or larger) systems and show that an attacker can quickly locate hidden regions of existing defenses with little or no trace.

Organization We introduce the threat model in Section 2. Section 3 provides the necessary background for our attacks, presented in Section 4. We then describe our methodology for discovering memory allocation primitives (Section 5) and evaluate their availability and the effectiveness of the proposed attacks in Section 7. Finally, we discuss the implications for the defense mechanisms that rely on ASLR for information hiding (Section 7.6), consider mitigations (Section 8), place our attacks in the context of related work (Section 9), and draw conclusions in Section 10.

2 Threat model and assumptions

The attacks presented in this paper apply to programs that contain vulnerabilities, but are, nevertheless, protected using state-of-the-art defenses. The sensitive data, vital for the correct operation of the defense, is isolated in a hidden region by means of information hiding. Hardware-based isolation, realized with segmentation on 32 bit x86 architectures, is not available. These assumptions correspond to some of the most advanced anti-exploitation defenses for x86-64 today [3]. Note that we assume an *ideal* information hiding implementation, i.e., all sensitive information is in a hidden region at a truly random location in a large virtual address space and the code that performs this pseudo-isolation, as well as the defense itself, contain no faults. In addition, we assume that the separation of sensitive and non-sensitive data is perfect; the process memory holds no references to the hidden region, and following pointers from non-sensitive regions can never lead to pointers into the hidden region.

We further assume an attacker with arbitrary memory read and write primitives. In other words, the attacker can read or write any byte in the virtual address space. However, we consider that all sensitive data, which could allow an attacker change the control flow of the program in order to execute arbitrary code, is hidden in the hidden region. Therefore, although the attacker can read any byte in memory, she cannot probe the address space by brute force without incurring program crashes or other noticeable events with high probability.

We assume that the target application runs on a modern Linux system with memory overcommit. This is a common configuration in many production systems, either because of the pervasive use of virtualization technologies [14], or because this is required or explicitly recommended for popular and complex services, Redis [15] and Hadoop [16] among others. We also generally consider (real-world) applications that either handle allocation failures appropriately or do not crash in a way that triggers a re-randomization (e.g., by forking and using `execve` to replace the worker process image) when the allocation request cannot be serviced. The goal of the attacker is to carefully utilize memory oracles to poke holes into the information hiding and reveal the location of the hidden region.

3 Background

In this section, we illustrate the organization of a typical process' virtual memory address space. While most of the discussion is based on Linux-based operating systems, we present fairly generic address space organization principles which apply to other systems as well. Understanding the memory layout of processes is vital for comprehending the mechanics of memory allocation oracles, detailed in the following sections.

Hole	Min	Max	Entropy[1]
A	130,044GiB	131,068GiB	28 bits
B	1GiB	1,028GiB	28 bits
C	4KiB	4GiB	20 bits

Table 1: Virtual memory hole ranges for a 64-bit position-independent executable (PIE) on Linux.

The default address space of a typical x86_64 position-independent executable (PIE) on Linux (kernel version 3.14.7 used as a reference) is depicted in Figure 1. The system randomly selects an address which serves as the starting offset of the process' `mmap` space. In kernel concepts, this is a per-address-space `mmap_base` variable. Shared objects, including the PIE executable itself, are allocated in this virtual memory-mapped area, which extends towards lower addresses. Figure 1 also illustrates several *holes* (unmapped regions) fragmenting the address space. Such holes have different purposes and semantics.

To support typical dynamic memory allocations, the process relies on a separate `[heap]` space, at the lowest level managed by `brk/sbrk` calls. As the stack grows down on x86, the heap is naturally designed to grow up towards the stack. The size of the hole between these two regions is randomized. The stack, in turn, is located at a random offset from the end of the user address space (i.e., at `0x7ffffffffffff`), giving rise to another variable-sized hole at the top.

To protect against trivial exploitation of NULL pointer dereferences by the kernel [17], processes are not allowed to map or access addresses ranging from zero up to an administrator-configurable limit (i.e., `vm.mmap_min_addr`, which defaults to 64KiB). Additionally, the small hole between the stack and VDSO is typically less than 2MiB. In less than 1% of the invocations, the VDSO object will end up either adjacent to the stack or adjacent to the linker object. In both cases, the layout is effectively the same, except that the small random hole may not be present.

In practice, the uncertainty in the layout of the address space is dominated by the sizes of the large hole from `vm.mmap_min_addr` to the end of the mmap space (hereafter referred to as hole *A*), the hole between the stack and heap (named *B*) and the hole covering the top of the user address space (named *C*). While there may be holes between the loaded shared objects, those are normally of a known (fixed) size. The sizes of these holes are all uniformly distributed in the ranges shown in Table 1.

[1] Calculated under the assumption that the distributions are indepen-

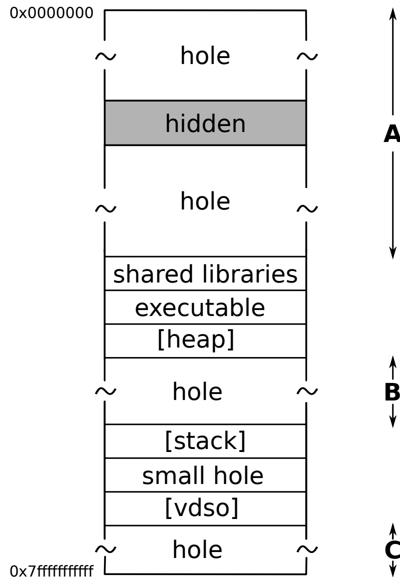

Figure 1: Virtual memory address space layout for a 64-bit position-independent executable (PIE) on Linux.

4 Memory Allocation Oracles

In this section, we thoroughly discuss the mechanics of two memory-allocation oracles, which can dramatically reduce the entropy of ASLR for accurately locating a hidden region in the virtual address space. The oracles can be realized through an *ephemeral allocation primitive* (EAP) and a *persistent allocation primitive* (PAP), respectively. Both primitives can be triggered by attacker-controlled input, say an HTTP request in a typical web server, and force a legitimate program path to allocate virtual memory with attacker-controlled size. By repeatedly using such primitives and monitoring the behavior of the target program (e.g., the error code in a HTTP response message), the attacker can infer the size of *holes* (unallocated space) in the virtual memory address space and learn key properties on its layout.

Whenever an EAP is used, the reserved virtual memory is released shortly after allocation (e.g., a short-lived per-request buffer), giving the attacker the opportunity to probe the target program multiple times. As detailed later, this allows an attacker to leak the size of the largest hole in the virtual memory address space and reduce the entropy of ASLR up to a *single* bit. The PAP, in turn, is based on reserving long-lived memory (e.g., a key-value store entry) and can be used in combination with the EAP to counteract the last bit of entropy or, by itself, to significantly reduce the entropy of ASLR.

dent. Any dependence naturally reduces the entropy.

4.1 Crafting primitives

The *ephemeral allocation primitive* (EAP) is available when a program allows *attacker-controlled input* to force the allocation of a short-lived memory object with an *attacker-controlled size*. In other words, the lifetime of the object must be such that the object is deallocated in a short amount of time (or by an attacker-controlled action, e.g. closing the connection that the object is associated with).

Even if an attacker induces the program to allocate a huge memory object of arbitrary size, such an allocation will succeed as large allocation operations typically result in an `mmap` system call. Thanks to demand paging, the system call returns right after reserving the required amount of virtual memory address space. The assignment of physical memory pages (page frames) to a virtual memory area, and even the population of the page tables, is only performed when a page is accessed for the first time. Hence, as long as the program does not immediately try to access all of the allocated virtual memory range, little physical memory is used and execution continues normally. This allows an attacker monitoring the program output to detect a *positive side effect* and verify that the corresponding address space was successfully reserved.

When the allocation size is larger than the amount of available virtual memory address space, however, such allocations will fail, typically causing the program to enter error-handling logic to return a particular error code (e.g., HTTP's 500) to the client. This allows an attacker monitoring the program output to detect a *negative side effect* and verify that the allocation failed.

Other than monitoring program behavior for side effects, the attacker needs to fulfill two requirements to craft an EAP. First, the attacker needs to find an input which induces the program to allocate a short-lived memory object. This is, in practice, straightforward, since most programs allocate objects as part of their input-handling logic and release them afterwards. Second, the attacker needs to coerce the program to use an attacker-controlled size for the target object. While exploiting a naive program neglecting to set limits on the resources it will reserve (for instance, on the buffer size per client) is an option, in many cases the size of an allocation is calculated based on long-lived values which are stored in memory. As a result, an attacker in our threat model can rely on an arbitrary memory write vulnerability to corrupt one of those values and effect a memory allocation of a chosen size to craft an EAP. An example is an attacker able to corrupt a `buff_size` or similar global variable to control the size of a target allocation instance, a very common scenario in practice. In later sections, we substantiate this claim with empirical evidence on real-

world applications and present a methodology that can assist an attacker in the fast discovery of our primitives (Section 5).

To craft a *persistent allocation primitive* (PAP), an attacker can similarly abuse allocation instances and corrupt allocation sizes. The only difference is that a PAP relies on long-lived memory objects whose lifetime is not under attacker control. For example, a server program maintaining long-lived memory objects in a cache (spanning across several input requests) is amenable to a PAP, provided the attacker can control the allocation size. Oftentimes, however, the attacker can leverage the same primitive to obtain both an ephemeral and a persistent allocation primitive. For example, the common case of attacker-controlled allocations associated with individual client connections allow an attacker to craft either an EAP (when using nonpersistent connections) and a PAP (when using persistent connections) in a fully controlled way.

4.2 Breaking IH using the EAP

Many modern defenses depend on *information hiding* (IH) in order to protect a sensitive area which contains, for example, code pointers. We now discuss how a crafted EAP can reveal the *hidden* area (or hidden object), with few or even zero program crashes [10] and other detection-prone events [11] (hereafter, simply "crashes"). We discuss how an attacker can hide her traces even further (certainty of no crashes) in the next subsection. Here, we describe a simplified attack assuming that the defense randomizes the location of the hidden object within the largest available memory region. This assumption is fairly often verified in practice, given that if all holes in the address space are uniformly considered for hosting the hidden object, the largest hole (A) is, on average, 261 times more likely to be selected than the second largest hole. We later lift this and other assumptions on the address space organization in Section 4.3.

Once the hidden object is created, the hole size (A) is split into two new hole sizes, a large (L) and a small (S) one. [2] Assuming a random placement of a hidden object of size H in A, the bounds of L and S are $L_{min} = (A_{min} - H)/2$, $L_{max} = A_{max}$, $S_{min} = 0$, $S_{max} = (A_{max} - H)/2$. Hence, the distributions for the sizes L and S overlap. However, since in any given instance $L > S$ and, assuming the hidden object H is reasonably smaller than A, L is now the largest hole in the address space. Hence, an attacker can quickly identify L using EAP-based binary search (formalized in Algorithm 1, Appendix A). In detail, at each binary search iteration, the attacker performs a single EAP invocation for a given allocation size and observes its positive or negative side effects to select the allocation size for the next iteration. When the search completes, the attacker learns the largest allocation size and thus L. There can never be any confusion while we are performing the binary search for L as, if an allocation cannot be satisfied from the larger hole, it can certainly not be satisfied by the smaller one.

Since the hidden object is equally likely to have been placed below or above the midpoint of A, there's is a 50% chance that L is the lower hole size. In this case, the location of the hidden object is precisely known: the base address of the hidden object is exactly located at `vm.mmap_min_addr`+L.

If L refers to the hole located higher than the hidden object, the uncertainty in the placement of the hidden object is the same as the uncertainty in the size of A. However, we can calculate the location of the hidden object based on the location of the `mmap` region. Given the interlinking of heap, stack, and code objects [18, 6, 1, 19], an attacker armed with an arbitrary memory read primitive can transitively explore allocated objects and discover the lowest memory mapped address. For example, in a typical quiescent application with a predictable memory layout, an attacker may simply leak `__libc_start_main`'s return address off the stack and immediately locate all the other virtual memory areas (VMAs) in the `mmap` region. Once the lowest memory mapped address `mmap_bottom` is known, the attacker can again deduce the location of the hidden object: its base address is exactly located at `mmap_bottom`-L.

Hence, the only uncertainty remaining is in the ordering of the two L- and S-sized holes, i.e., a *single* bit of entropy. In other words, an attacker probing the address space with an arbitrary memory read primitive has a 50% chance of discovering the hidden object on the first try and a 100% chance if she can tolerate a single invalid memory access. Even for nonforking server programs, if a process eventually gets restarted (either manually or automatically) with different randomization, the attacker has a 75% chance of guessing the hidden object's location correctly after one crash, 87.5% with two crashes, 93.75% with three crashes and so on. While this attack is already fast and stealthy with great chances of going unnoticed in most practical settings, we show how to improve it even further without a single crash in the next subsection.

4.3 Using both the EAP and the PAP

When, in addition to the EAP, the attacker is in a position to employ a PAP as well (as it is often the case in practice), she can reliably break information hiding with no application crashes. In addition, she can locate the hidden object regardless of the original hole it was ran-

[2] For brevity, we omit explicit discussion of the case where $L = S$, which does not deviate from the common case where $L > S$.

domly placed in (lifting our original assumptions). For simplicity, let us first consider the case of a hidden object placed somewhere in the middle of A, such that L and S are the first and second largest hole sizes in the address space. To recover L, the attacker begins by executing the EAP-only attack in Section 4.2. Subsequently, she can simply use the PAP to perform a L-sized allocation and eliminate the largest (L-sized) hole from the process' address space. Finally, she can repeat the same EAP+PAP strategy on the now largest allocation size in the address space to recover S and fill the remaining (S-sized) hole.

Since A is now completely hole-free, an attacker armed with an arbitrary memory primitive can reliably probe for the hidden object in the two possible locations in A and eliminate the remaining uncertainty. In detail, if the L-sized hole was at the beginning of the address space (and has now been filled by the PAP allocation), a read from `vm.mmap_min_addr`+S will be accessing zero-filled pages. If the S-sized hole was at the beginning of the address space, in turn, a read from `vm.mmap_min_addr`+S will be accessing pages containing data from the hidden object. In either case, by combining the EAP and the PAP, the attacker can easily disclose the location of the hidden object with no risk of crashes, quickly and stealthily exhausting information hiding's entropy.

Let us now reconsider our original assumption. In the general (if unlikely, for practical reasons [3]) case, the hidden object might be placed in a hole other than A. However, this is hardly a problem for an attacker armed with both the EAP and PAP. Such a zero-knowledge attacker can simply start with a single iteration of the EAP+PAP attack to fill the largest hole in the address space, then move to the second largest, and so on, until she can infer enough knowledge to first locate the hidden object's owning hole and then its location. For example, if the first largest possible hole identified is `sizeof(A)`, the attacker can learn the object is placed in either B or C. If the second largest possible hole identified is `sizeof(B)`, the attacker can learn the object is placed in C. At that point, she can perform the A-style EAP+PAP attack introduced earlier and locate the object with no crashes.

We note that exhausting the virtual address space with our iterative EAP+PAP attack strategy is not a concern in real-world scenarios. First, legitimate program allocations are normally satisfied by allocator arenas which rarely need to be extended during steady-state operations. In addition, the location of a target hidden object can in practice be determined without exhausting all the available holes. For example, an attacker could infer the location of a hidden object in A by only filling the L-sized hole and reliably reading from `vm.mmap_min_addr`+L.

[3]Placing the hidden object between stack and heap may impose unexpected limits on the growth of an application's data.

4.4 Using only the PAP

When allocations have to be persistent (e.g., only the PAP is available or the EAP has less desirable side effects), there are two main difficulties. First, given what we know about the hole size distributions, there might be multiple holes which can satisfy a request, but, without knowing their actual sizes, we cannot always tell which hole an allocation came from. Second, when an allocation succeeds, even if we know which hole it came from, we learn that that hole is at least as large. Contrary to the EAP though, we cannot retry a larger allocation size since we cannot "undo" the allocation.

An example serves to demonstrate. Suppose we start with the typical layout of a PIE executable (Table 1). Let us say we attempt an allocation of 130,500GiB and the allocation succeeds. The allocation was necessarily satisfied from hole A. We now have a lower bound on the size of A, yet we would like to find out its exact size. However, if we try to allocate a value in the range of 0-568GiB and the allocation succeeds, we cannot know whether the space was reserved in hole A or hole B as their size distributions now overlap.

We have designed and implemented a novel attack strategy which significantly reduces the uncertainty in the sizes of the holes. Our algorithm tracks the maximum allocatable size, as well as the allocated size, for each hole in what constitutes a *state*. Our approach then relies on two insights. First, it is highly preferential to probe using allocation sizes that can only be satisfied by a single hole. Second, when forced to perform an allocation which could have been satisfied from more than one hole, we need to fork and keep track of multiple states to model each feasible configuration of the holes in the targeted address space.

Building on these insights, our algorithm follows a cost-driven strategy to allow an attacker to select an optimal tradeoff between the number of allocation attempts and the entropy reduction obtained. We quantify this tradeoff in Section 7.5 and refer the interested reader to Appendix B for a detailed walkthrough of the formalized algorithm.

4.5 A more powerful EAP-only attack

Section 4.2 detailed how to locate the hidden object when it was placed in the largest address space hole (A). We then lifted this restriction by making use of the PAP in Section 4.3. An alternative way of stealthily probing for holes other than the largest one using the EAP only (when no PAP is available), is to try and trigger more than one EAP simultaneously. After having recovered L (Section 4.2), the attacker can simultaneously issue an allocation of L bytes while using a different allocation

request to probe for S. Even if the window is small, repeated simultaneous requests can make the chance of a false allocation arbitrarily small. When the program can afford more EAPs to be issued in parallel, the attack further improves, as the L-sized hole can be kept filled more reliably while a binary search is running to determine S. This approach generalizes to any number of holes.

4.6 Handling internal allocations

The attacks detailed in this section consider allocation primitives directly or indirectly based on `mmap`. However, when the primitive interacts with standard glibc allocation functions (e.g., `malloc`, `calloc`, `posix_memalign`, etc.), the result is one internal allocation for exceedingly large requests. Even though the requested size clearly does not fit in the largest available hole, glibc (version 2.19) allocates a new heap arena. The heap arena is allocated in the memory-mapped space and it is 64MiB-aligned. Therefore, the actual size of A that is recovered by a binary search differs from the previous end of the memory-mapped area by a random number which is 14 bits wide ($2^{26}/2^{12} = 2^{14}$).

Nevertheless, this is not a problem in practice, as the heap arenas form a circular, singly-linked, list. Therefore, an attacker armed with an arbitrary memory read primitive can navigate the links from the main allocation arena and discover all arenas in use (typically there would only be one link to follow). The main arena is a static variable in glibc, so it is located at a known (binary-dependent) offset from the highest address of the `mmap` space. Hence, as soon as the attacker leaks a pointer into the `mmap` space, she can easily account for the newly allocated heap arena as well.

5 Discovering Primitives

So far, in Section 4, we have discussed the mechanics of ephemeral and persistent memory-allocation primitives, which can assist an attacker in revealing the allocated ranges of a process in the virtual address space. In this section, we show that *dynamic data-flow tracking* (DFT) techniques applied to popular server programs can effectively assist attackers in discovering allocation instances that can potentially be abused to craft our primitives.

Discovering primitives that can result in powerful memory-allocation oracles involves identifying memory locations that, once controlled, can influence the input parameters of memory-allocation functions. Recall that, from Section 2, we assume attackers that are already in possession of (at least) one arbitrary read and write primitive. What the attacker lacks is a methodology to guide her to apply the read/write primitives and successfully craft EAPs and PAPs.

To model an attacker with arbitrary read/write control over memory, we start our analysis from a *quiescent state* of the program under attacker control. This state can be also manipulated and exercised over time by the attacker. As a simple example, consider a vulnerable web server. Assume the attacker can send a first a special-crafted HTTP request to trigger an arbitrary memory write vulnerability and gain control over memory. Next, the attacker issues a second request to invoke a memory allocation oracle. Therefore, in this particular example, our attacker-controlled quiescent state corresponds to that of an idle web server waiting for new requests. Once the second request is served, many parts of memory can be influenced either explicitly or implicitly. At one point, processing of the request triggers some memory-allocation function which serves as an oracle. It is important to stress that, depending on the server's logic, parts of memory are overwritten (through successive allocations), while the request is processed. These parts cannot be generally controlled by the attacker using her arbitrary read/write primitives. However, the attacker *still* controls all the memory which was available in the original quiescent state (before the second request takes place). As long as memory of that state reaches a memory-allocation function, then the attacker can successfully use the oracle. Therefore, what we need to determine is the memory locations that influence memory-allocation sites and are *still* attacker-controlled, once the second HTTP request triggers memory-allocation functions.

Practically, this model can be easily realized using DFT. For our purposes, we use Memory-allocation Primitive Scanner (MAPScanner), a custom scanner based on libdft [20]. We start an application instrumented by MAPScanner, with all memory untainted, with *no* taint sources, and with all memory-allocation functions defined as sinks. Once the application is idle, we signal MAPScanner to taint all memory. At this point, the target quiescent state is defined and we assume that all memory is attacker-controlled. We then proceed and send a request to the server application. Any subsequent memory allocations that are triggered by the second request, since we have defined *zero* taint sources, wash out the taint of previous attacker-controlled memory. While the request is processed, MAPScanner reports all memory-allocation functions which are initiated with input from *still*-tainted, and therefore, *still*-controlled memory.

Notice, that, depending on the selected quiescent state and the input request, the attacker can discover more or fewer primitives. Using several complicated quiescent states, for example, those between handling two successive requests or between accepting the socket and receiving data, may uncover additional primitive candidates.

Once primitive instances are found, the attacker sim-

ply needs to locate the controlling data in memory (often a `buff_size` global variable originating from the configuration file), corrupt the data (and thus the allocation size) with an arbitrary memory write primitive, and monitor the execution for side effects. To classify a potential primitive as an EAP or PAP, the attacker will need to use the source or runtime experimentation to determine the lifetime of the corresponding allocated object along different program paths. Further, manual investigation is required to eliminate primitives that might not be usable because the value in memory is subject to additional validity checks in the attacker controlled paths.

In practice, we found that even when selecting the simplest quiescent state (i.e., idle server) and input (i.e., simple client request), an attacker can locate sufficient usable primitives to mount our end-to-end attacks (see Section 7).

6 Exploiting Timing Side Channels

Not all discovered primitives may automatically guarantee a realistic and crash-free attack. Certain types of primitives may not have any directly observable side effects (e.g., the server transparently recovering from allocation failures), making exploitation more complicated. Other types of primitives may result in program crashes (e.g., the server failing internal consistency checks), typically in both successful and unsuccessful cases, again making it difficult for an attacker to distinguish the two cases via direct observation. In both scenarios, however, an attacker can still infer the allocation behavior (success or failure) by measuring the time it takes to handle every particular request. We exemplify timing attack strategies for both the imperfect primitives presented above.

Even when a primitive has no directly observable side effects, allocation of memory and failure to allocate memory normally take a different amount of time. On Linux, for instance, a successful allocation is typically satisfied by a small VMA cache, avoiding lengthy walks of the red-black tree of virtual memory area (VMA) structures. However, on a VMA cache miss, before declaring an allocation failure, the kernel needs to walk all the nodes in the red-black tree in a compute-intensive loop, which takes measurably longer time to complete generating a *timing side channel* [21]. In fact, many kernel optimizations, such as VMA merging [22], explicitly seek to reduce the run-time impact of such expensive red-black tree walks. The timing signal becomes stronger for programs maintaining many VMAs and much stronger if the attacker can lure the program into allocating even more VMAs (however, VMA merging normally makes this difficult even for a PAP-enabled attacker). Even stronger timing side channels may be generated by the program itself. For example, to transparently recover from allocation failures, the program may employ complex and time-consuming error-handling logic or log the event to persistent storage.

When a primitive results in program crashes in successful and unsuccessful cases, in turn, the presence and the strength of timing side channels is entirely subject to the internal cause of the crash. Interestingly, we found that the leading cause of crashes results in a very strong side channel. In fact, successful allocation-induced crashes are most commonly induced by a server attempting to fully initialize (or access) the huge allocated block, resulting in several time-consuming page faults before leading to the final out-of-memory error. As shown in Section 7, timing attacks which rely on crashes are remarkably effective in practice.

7 Evaluation

7.1 Primitive Discovery Results

We apply MAPScanner to a variety of well-known and popular server software. In particular, we consider BIND 9.9.3 (a DNS server), lighttpd 1.4.37 and nginx 1.6.2 (two popular web servers), as well as mysql 5.1.65 (a widely deployed database server). We built all programs using their default options (i.e. optimizations were enabled).

Since the presented applications have the form of a server accepting and servicing requests, we select, as the (simplest possible) attacker-controlled quiescent state, the point when the server is idle waiting for incoming connections, and, as the (simplest possible) attacker-controlled input, a default request to the server (Section 5). Of course, motivated attackers can carry out similar analyses starting from several additional quiescent states and inputs, so our results here are actually an (already sufficient) underapproximation of the real-world attack surface. Notice, finally, that we assume each server is being protected by an information-hiding-based defense mechanism which thwarts direct exploitation attempts (e.g., control-flow diversion).

Table 2 presents all the primitives discovered by our analysis. We name each instance of a primitive after the variable that an attacker needs to corrupt in memory to craft the corresponding allocation oracle. For each of the primitives, we report the type width of the memory-resident value that influences the allocation site. While 32-bit fields are only sufficient to bypass 32-bit (and not 64-bit) information hiding, we believe their availability can be indicative of the risks for 64-bit defense mechanisms—e.g., code refactoring changing an allocation size type to the common 64-bit `size_t` type may inadvertently introduce allocation oracles.

Table 2: For each particular application, we report the number of primitives found, the width of the allocation value, whether the primitive forces a crash, whether timing is necessary to determine success and if the primitive can be persistent as well. The "RE" (Residual Entropy) column assumes an attacker can reliably exploit the associated timing side channels. Values marked with (*) refer to lighttpd configured in forking mode.

	Primitive	Size	Crash-free	Timing-dependent	EAP	PAP	RE (bits)
bind	mgr->bpool	64-bit	✗	✓	✓	✗	1
	heap->size	32-bit	✓	✗	✓	Primarily	0
lighttpd	buffer->size#1	64-bit	✗	✓	✓	✗	0*
	buffer->size#2	64-bit	✗	✓	✓	✗	0*
	config_context->used	64-bit	✗	✓	✓	✗	0*
nginx	ls->pool_size	64-bit	✓	✗	✓	✓	0
	client_header_buffer_size	64-bit	✓	✗	✓	✗	1
	request_pool_size	64-bit	✓	✗	✓	✗	1
mysql	net->max_packet	32-bit	✓	✗	✓	✓	0
	net_buffer_length	32-bit	✓	✗	✓	✓	0
	connection_attrib	64-bit	✓	✗	✓	✓	0
	query_prealloc_size	64-bit	✓	✓	✓	✓	1
	records_in_block	32-bit	✓	✗	✓	✗	1

Additionally, we checked whether utilizing a primitive carried a risk of crashes ("crash-free" column). For primitives that did not provide directly observable side effects, the "timing-dependent" column indicates that the attacker needs to conduct a timing side channel attack to craft her primitives (we provide an example in Section 7.3). The EAP and PAP columns specify that the primitive can be used to perform an ephemeral and persistent allocation (respectively). Finally, we quantify the residual entropy after we perform the best attack at the attacker's disposal for each primitive.

For each of these applications, our simple methodology was sufficient to discover 64-bit primitives able to quickly locate hidden objects with no residual entropy. In most cases, the discovered primitives were crash-free and could function as both EAPs and PAPs.

nginx and mysql are the best examples. They both provide ideal EAP+PAP attack primitives to stealthily bypass 64-bit information hiding with little effort. It is also worth noting that the `connection_attrib` primitive in mysql involves overwriting the requested stack size in a `pthread_attr_t` struct. As such, we expect a similar primitive to be available in all servers that create threads to service clients (either overwriting an application-specific attribute structure or the one in glibc).

For lighttpd, the server's default configuration only allows the EAP-only attack, but, when the server is configured with forked worker processes, an attacker can successfully conduct the side-channel attack exemplified in Section 7.3 to eliminate all entropy and bypass information hiding.

Bind stands out as, depending on the server configuration, the `heap->size` primitive might be usable as an EAP or may effectively only function as a PAP. The reason for this behavior is that the effected allocation becomes part of a relatively long-lived cache. Hence, its lifetime is determined by administrator choices and performance considerations. When cached objects are not eagerly expired, the primitive may only be usable as a PAP for the duration of a practical attack.

Overall, our simple analysis shows that real-world information-hiding-protected applications stand very little chance against attackers armed with allocation oracles.

7.2 EAP+PAP attack on nginx

To illustrate how the combined EAP+PAP attack works in practice, we consider the `ls->pool_size` primitive discovered during our investigation of the nginx web server (Table 2).

When servicing a new connection, nginx's `ngx_event_accept()` function allocates a per-connection memory pool (`c->pool`) using the size stored in the listening socket data structure associated to the socket the `accept()` originated from. `ngx_event_accept()` instantiates the pool by calling out to `ngx_create_pool()`, which eventually allocates the required memory by means of `posix_memalign()`.

Using our primitive discovery methodology, we were easily able to determine that the `size` argument to `posix_memalign()` originated from a value resident in live memory for our idle attacker-controlled quiescent state. This means that an arbitrary memory write vulnerability in any of the code that processes untrusted in-

put can be used to overwrite this value with an attacker-selected size, once the memory location is known.

We then verified that `ls->pool_size` is trivially accessible by following the `ls` field of the `ngx_connection_t` structure, a pointer to which is always available on various stack locations while the server is executing request-processing code.

Using this information, the attacker is able to craft an ephemeral allocation primitive by using an arbitrary read to navigate the pointer chain until she determines the address of `ls->pool_size`. At this point, she can effect a call to `posix_memalign()` with a size of her choosing by overwriting `ls->pool_size` and then opening a connection to the server. If the allocation request was successful, the attacker can issue an HTTP request over that new connection (positive side effect). If the allocation cannot be accommodated, the connection is forcibly closed by the server (negative side effect).

Using the same procedure, the attacker can craft a PAP by simply keeping the connection open in the last step. To conduct the complete attack, the attacker first employs the EAP to determine the size of the larger of the two holes around the hidden object (for simplicity, we only discuss the case when the hidden object is placed in the largest contiguous pre-existing hole; other scenarios are investigated in Section 4.3). Having determined the maximum allocation (i.e., hole) size, she relies on the PAP to allocate the exact size of the larger hole, taking it out of the picture. She then proceeds to conduct the EAP-based attack against the smaller hole around *S*. Finally, she simply probes at address `vm.mmap_min_addr`+*S* to complete the attack, as described in Section 4.3.

7.3 Timing-based attack on lighttpd

Next, we focus on the execution of an EAP-only attack which relies on a timing side channel. To demonstrate such an attack, we rely on the `config_context->used` primitive in lighttpd. In order craft this primitive, we configured lighttpd to use worker processes by setting the `server.max-worker` configuration variable to a non-zero value. With no loss of generality, we limit our analysis to one worker process, as an arbitrary memory access primitive makes it a matter of book-keeping to tag the workers (e.g. by writing a different value for each worker to an unused memory location), so that the attack code can target a single process.

Again using our primitive discovery methodology, we easily determined that `srv->config_context->used` is used as an argument to `calloc()` in the body of `connection_init()`. Similarly, we showed that pointers to `srv` are available in the stack frames above the event loop, which renders `srv->config_context->used` trivially accessible to an attacker equipped with arbitrary memory read/write primitives.

The second argument to `calloc()` at this call site is `sizeof(cond_cache_t)`, which amounts to 144 bytes. Since that is less than the default page size on x86, we can always find a value that will result in the allocation of any given number of pages.

Crucially, the return value from `calloc()` is never checked for failure. Therefore, the only way to determine whether the allocation succeeded or not is to send a simple request so as to drive the server to a path which will dereference the pointer. That path is simply `http_response_prepare() -> config_cond_cache_reset()`, which will iterate over all elements of the array. As a result, if the allocation fails, the worker process *immediately* crashes on trying to access the first element, resulting in a closed connection for the client. If the allocation succeeds but the allocated size is much larger than the amount of physical memory on the system, this allocation incurs *several lengthy page faults* before causing an out-of-memory (OOM) condition—on which Linux' "OOM killer" terminates, with high reliability, the worker process. If the system can survive faulting in all the allocated pages (presumably because the allocation was "small"), the server eventually sends back a response. Using either the timing or the reception of an HTTP response, we can infer whether the attempted allocation succeeded or failed.

When the worker process crashes or is terminated by the OOM killer, the parent is notified and forks a new child in replacement (indeed one of the motivations for using worker processes in server software is for crash recovery purposes). Each new worker process inherits the address space of the parent; hence, all memory regions (including the hidden object) remain at stable virtual addresses across worker restart events.

Given the several page faults incurred before a crash, the timing side channel we rely on yields a very strong signal. In our testing, we performed the attack 40 times and were able to reliably differentiate between a successful and a failed allocation in all of them. By using the same primitive as a PAP (as is possible in this configuration), we were able to persistently allocate the recovered size for the largest hole and then repeat the EAP attack on the smaller hole as done earlier. In summary, by relying on EAP+PAP primitives and a strong timing side channel we could successful recover the address of the hidden object and bypass information hiding in all cases in our experiments.

7.4 EAP-only attack

We evaluated the accuracy and performance of the EAP in defeating the information-hiding properties of ASLR

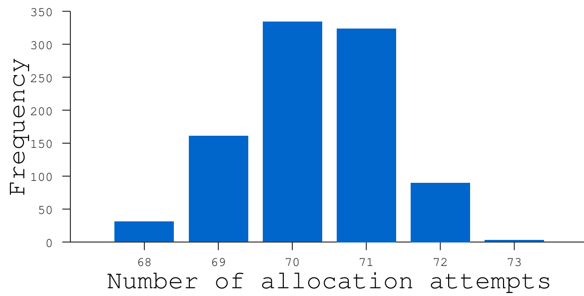

Figure 2: Histogram of the number of operations for recovering the exact hole sizes around a hidden object in an nginx executable [n=1000]

by preallocating a hidden object of a size of 2MiB in the address space of nginx (compiled as a position-independent executable) and then trying to determine the sizes of the larger and smaller holes on either side of it, as described in Section 7.2. Taking into account the complications and workarounds described in Section 4.6, we were able to exactly determine the size of both the larger and smaller hole and subsequently uncover the exact location of the hidden object, without incurring any invalid memory accesses.

Figure 2 depicts the number of required allocation attempts over 1000 runs (using different random configurations). On localhost and using gdb to effect the arbitrary memory access, the attack completed after an average of 28.20s with a median of 28.21s.

7.5 PAP-only attack

When the only primitive available to the attacker is the PAP, she needs to consider a number of tradeoffs. Clearly, the attacker is interested in reducing her entropy with respect to the position of the hidden region in the targeted address space. At the same time, different considerations might cause her to strive for minimal or rapid interaction with the target process. For example, a very large number of requests to a remote server might very well increase the chance that the attack will be noticed by network intrusion detection systems. Similarly, as the duration of the attack increases, so does the chance that unrelated process activities, such as servicing requests for other clients or periodically scheduled work, may interfere with the workings of the algorithm.

There exist two tunable parameters that affect the behavior of our PAP-only attack. One selects between the number of allocation attempts and the entropy reduction obtained, the other between entropy reduction and risk of failure.

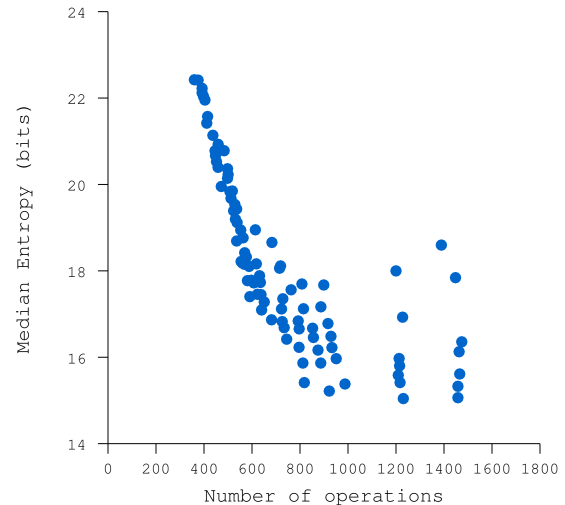

Figure 3: Parameter space exploration for the PAP-only attack

Figure 3 presents the number of allocation attempts versus the *median* of the residual entropy in the size of the largest hole after the completion of the attack, for every pair of parameter values that we explored. We can observe that extreme choices of the parameter values result in pathological behavior, either in the number of operations or in the residual entropy or both.

Conversely, there exist better parameter choices which do reasonably well for both metrics. Appendix C goes into more depth on the parameter values and their effect on the behavior of the attack.

Observe that, in attack scenarios where a number of requests on the order of a thousand is acceptable, there exist several parameters for which the median residual entropy is reduced to 15 bits. Notice also that when the size of the hidden region can be expected to be comparable to the residual entropy, the chances of successfully probing for the region are significantly increased.

For a round-trip time of 1s, even the PAP-only attack would take at most half an hour (12 minutes on average), which is still an eminently practical amount of time in many real-world settings.

7.6 Effectiveness against modern defenses

The presented attacks change the picture for the protection offered by state-of-the-art defense mechanisms that rely on information hiding. CPI's safe area [3] and many other prior solutions [4, 23, 19, 24] rely on information hiding to protect a single hidden region. For all such solutions, our attacks in Section 4 apply directly and can

locate the hidden region with little or no crashes. Prior work has demonstrated a memory probing-based crash-free CPI bypass with roughly 110,000,000 operations on nginx [12]. Using our combined EAP and PAP crash-free attack, we can replicate their attack needing only 74 operations in the worst case (Fig. 2). This is a 1,400,000x improvement in attack efficiency, which, projected on the request time reported in [12], translates to 0.23s (rather than 97 hours) to locate the hidden region. In addition, our crash-free attack is even faster than the fast crash-prone attack presented in [12] (6s with 13 crashes).

More recent client-side probing attacks [11] offer similar guarantees (i.e., locating CPI's safe area in 32 probes), but their probing strategy relies on exception handling rather than crash recovery, ultimately improving the attack efficiency. We note that both existing probing attacks [12, 11] exploit assumptions on CPI's huge hidden region size (on the order of 2^{42} bytes when using a sparse table and $2^{30.4}$ when utilizing a hash table [11]) to reduce the entropy and make the attack practical. In stark contrast, our attacks make no assumptions on the region size, and doing so would allow even a PAP-only attack to succeed without crashes.

Other solutions, such as ASLR-Guard [6], SafeStack [3], and other shadow stack implementations [25], rely on information hiding to protect multiple hidden regions. For example, all the shadow stack solutions need to maintain a per-thread hidden region. We note that our attacks generalize to multi-region information hiding with essentially the same impact. In particular, while multi-threaded programs disqualify the simple EAP-only attack, our best (EAP+PAP) attack naturally extends to multi-region solutions and can quickly bypass them (although more allocations may be required).

Finally, many leakage-resilient defenses [26, 27, 28, 29] enforce execute-only memory to protect the hidden (code) region from read-based disclosure attacks [1]. However, such defenses are susceptible to execution-based disclosure attacks in crash-tolerant applications [10]. To counter such attacks, some solutions deploy booby traps in out-of-band trampolines [26, 27]. With allocation oracles, an attacker can sidestep the booby-trapped trampolines and quickly find the hidden region, enabling more practical and guided execution-based disclosure attacks against such defenses.

8 Mitigations

One strategy to defend against allocation oracles is to enforce an upper limit on the maximum amount of virtual memory that a process can allocate. This mechanism is already available on Linux (and other POSIX-compatible operating systems) via the `RLIMIT_AS` resource limit (adjustable via `setrlimit`). Setting this limit to a small, though still sufficient for most current applications, value would thwart any attempts to probe the sizes of the larger holes in the address space. The resource limit can be set by the application itself (in which case, a defense mechanism could intercept it and adjust it to accommodate its own needs for virtual addresses) or it can be hard-capped by the administrators without any need for program adjustments. The main difficulty lies in predicting the maximum virtual address space usage under all conceivable conditions so as to never deny legitimate allocation requests for production applications. Nonetheless, its wide availability, straightforward deployment, and robustness (see below) make `RLIMIT_AS` our primary recommendation for compatible workloads and configurations.

For some classes of applications (especially those relying on memory overcommit), limiting the amount of virtual address space available may be problematic, e.g., when memory-mapping huge files. In such cases, one could switch to a strict overcommit policy and have the applications always use `mmap`'s `MAP_NORESERVE` flag for huge—but known to be benign—allocations. `MAP_NORESERVE` instructs the kernel not to count the corresponding allocations towards the overcommit limit. However, this mitigation strategy would still allow an attacker to inject the `MAP_NORESERVE` flag in `mmap` calls using memory-resident arguments and craft our primitives. Another issue with such a strict overcommit strategy is that it is incompatible with memory-hungry applications that rely on `fork` and cannot simply switch to `vfork` (the redis server being a prime example). This problem can only be directly mitigated with the addition of a new flag to the `clone` system call to mimic the semantics of `MAP_NORESERVE`.

In some setups, one may deploy an IDS looking for anomalous events (i.e., allocations) in a given application [30]. However, this approach generally requires per-application policies (e.g., only allow huge allocations of a specific size). A policy looking for frequent huge allocations in arbitrary applications is more generic but problematic, as an attacker can easily dilute the very few probing attempts required by our attacks over time [30].

Finally, defense mechanisms could bracket their hidden regions with randomly-sized trip hazard areas [31] to deter in-region memory probing. This is the immediate systemwide mitigation we recommend for information-hiding-based solutions already deployed in production [32]. Albeit still probabilistic (and thus prone to attacks), such solution can also provide efficient protection against other (known) side-channel attacks [31].

9 Related work

We distinguish between approaches that that aim at breaking ASLR in general and approaches that try to

break more advanced defense techniques that rely on ASLR-based information hiding.

Breaking ASLR has been fertile research ground for years and became especially popular in recent years. From the outset [30], pioneering work showed that the randomization in 32-bit address spaces provide insufficient entropy against practical brute-force attacks, so we focus on 64-bit architectures (x86-64) in this section.

In practice, bypassing standard (i.e., coarse-grained, user-level) ASLR implementations is now common. For an attacker, it is, for instance, sufficient to disclose a single code pointer to de-randomize the address space [33]. Even fine-grained ASLR implementations [34] are vulnerable to attacks that start with a memory disclosure and then assemble payloads in a just-in-time fashion [1].

More advanced attack vectors rely on side channels via shared caches. Specifically, recently accessed memory locations remain in the last-level cache (LLC) which is shared by different cores on modern x86-64 processors. As it is much faster to access memory locations from the cache rather than from memory, it is possible to use this timing difference to create a side channel and disclose sensitive information. By performing three types of `PRIME+PROBE` attacks on the CPU caches and the TLB, Hund et al. [35] could completely break kernel-level ASLR by mapping the entire virtual address-space of a running Windows kernel. To perform a `PRIME+PROBE` attack, the attacker needs the mapping of memory locations to cache sets. In modern Intel processors, this mapping is complex and reverse engineering requires substantial effort [35]. However, performance counter-based and other techniques have been proposed to lower the reverse engineering effort [36].

Even without *a priori* disclosures, attackers may still break ASLR using Blind ROP (BROP) [10]. A BROP attack sends data that causes a control transfer to another address and observes the behavior of the program. By carefully monitoring server program crashes, hangs, or regular output, the attacker can infer what code executed and, eventually, identify ROP gadgets. After many probes (and crashes), she gets enough gadgets for a ROP chain. BROP is a remote attack method applicable (only) to servers that automatically respawn upon a crash.

In general, leaking information by means of side channels is often possible. To launch such an attack, an attacker typically uses memory corruption to put a program in a state that allows her to infer memory contents via timings [21, 37, 12] or other side channels [10].

As ASLR by itself does not provide sufficient protection against the attacks described above, the community is shifting to more advanced defenses that build on ALSR to hide sensitive data (such as code pointers) in a hidden region in a large address space, typically not referenced by any pointers within the attacker's reach.

Hiding secret information in a large address space is now common practice in a score of new defenses. For example, Oxymoron [4] protects the *Randomization-agnostic Translation Table* (RaTTle) by means of information hiding, and Opaque CFI [23] protects the so-called *Bounds Lookup Table* (BLT) in a similar way. Likewise, Isomeron [19] keeps the *execution diversifier data* secret and StackArmor [25] isolates potentially vulnerable stack frames by means of hiding in a large address space. Finally, on x86-64 architectures, CFCI [24] also needs to hide a few MBs of protected memory.

One of the best-known examples of a defense that builds on ASLR-backed information hiding is Code Pointer Integrity [3]. CPI splits the address space in a standard and a safe region and stores all code pointers in the latter, while restricting accesses to the (huge) safe region to CPI-intrinsic instructions. Moreover, it also provides every thread with a shadow stack (called SafeStack in CPI) in addition to the regular stack and uses the former to store return addresses and other proven-safe objects. Both the shadow stacks (which are relatively small) and the safe region (which is huge) are hidden at a random location in the virtual address space.

By means of probing on a timing side channel, Evans et al. showed that it is possible to circumvent CPI and find the safe region [12]. However, depending on the construction of the safe region, the attack may require a few crashes or complete in several hours in order to be stealthy (i.e., crash-free). Moreover, similar to the recent CROP [11] (which instead relies on specially crafted crash-resistant primitives), this attack needs to resort to full memory probing to locate small hidden regions (unlike CPI's) in absence of implementation flaws. Full memory probing forces the attacker to trigger many crashes and other detection-prone events, and its efficiency quickly degrades when increasing the address space entropy.

Concurrent work [31] relies on *thread spraying* to reduce the entropy in finding a per-thread hidden object. Allocation oracles can make thread spraying attacks faster by providing a more efficient disclosure primitive compared to the memory probing primitives used in [31].

Unlike all the existing attacks, *allocation oracles* demonstrate that an attacker can craft pervasively available primitives and locate the smallest hidden regions in the largest address spaces, while leaving little or no detectable traces behind.

10 Conclusions

We have shown that information hiding techniques that rely on randomization to bury a small region of sensitive information in a huge address space are not safe on modern Linux systems. Specifically, we introduced new in-

formation disclosure primitives, allocation oracles, that allow attackers to probe the holes in the address space: by repeated allocations of large chunks of memory, the attacker discovers the sizes of the largest areas of unallocated memory. Knowing the sizes of the largest holes greatly reduces the entropy of randomization-based information hiding and allows an attacker to infer the location of the hidden region with few to no crashes or noticeable side-effects. We have also shown that allocation oracles are pervasive in real-world software.

Unfortunately, information hiding underpins many of the most advanced defense mechanisms today. Without proper mitigation, they are all vulnerable to our attacks. While one may deploy more conservative memory management polices to limit the damage, we emphasize that the problem is fundamental in the sense that allocation oracles always reduce the randomization entropy, regardless of the mitigation and the address space size. In general, information hiding is vulnerable to entropy reduction by whatever means and it is not unlikely that attackers can combine allocation oracles with other techniques. In our view, it is time to reconsider our dependency on the pseudo-isolation offered by randomization and opt instead for stronger isolation solutions like software fault isolation or hardware protection.

11 Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

12 Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571 and by Netherlands Organisation for Scientific Research through project NWO 639.023.309 VICI "Dowsing".

References

[1] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE S&P '13*.

[2] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ASIACCS '15*.

[3] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI' 14*.

[4] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX Security '14*.

[5] PaX Team, "Address space layout randomization (ASLR)," 2003, http://pax.grsecurity.net/docs/aslr.txt.

[6] 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 *CCS '15*.

[7] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *SOSP '93*.

[8] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc, "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE S&P '07*.

[9] L. Deng, Q. Zeng, and Y. Liu, "ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries," in *IFIP SEC '15*, 2015.

[10] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *IEEE S&P '14*.

[11] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding," in *NDSS '16*.

[12] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity."

[13] "CVE-2015-3864," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3864.

[14] D. Magenheimer, "Memory overcommit... without the commitment," in *Xen Summit*, 2008.

[15] "Redis administration," https://web.archive.org/web/20150905213905/http://redis.io/topics/admin.

[16] E. Sammer, *Hadoop Operations*, 2012, ch. 4.

[17] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against return-to-user attacks," in *USENIX Security '12*.

[18] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *CCS '15*.

[19] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *NDSS '15*.

[20] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *VEE '12*.

[21] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *CCS '14*.

[22] "Mmap speedup," http://www.verycomputer.com/180_d89089d5a857ed08_1.htm.

[23] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *NDSS '15*.

[24] M. Zhang and R. Sekar, "Control-flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks," in *ACSAC '15*.

[25] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *NDSS '15*.

[26] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *IEEE S&P '15*.

[27] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table randomization and protection against function-reuse attacks," in *CCS '15*.

[28] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *CODASPY '15*.

[29] C. L. Kjell Braden, Lucas Davi and M. F. P. L. Ahmad-Reza Sadeghi, Stephen Crane, "Leakage-resilient layout randomization for mobile devices," in *NDSS '16*.

[30] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS '04*.

[31] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *USENIX Security '16*.

[32] "SafeStack," http://clang.llvm.org/docs/SafeStack.html.

[33] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *EuroSec '09*.

[34] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Sec '12*.

[35] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE S&P '13*.

[36] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *RAID '15*.

[37] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *IEEE S&P '16*.

A Base EAP-only algorithm

Algorithm 1 Binary search using the Ephemeral Allocation Primitive; sizes are in pages.

```
function DEDUCE(low, high)
    if low = high then
        return low
    if high - low = 1 then
        res ← TEMP-ALLOC(high)
        if SUCCESS(res) then
            return high
        else
            return low
    midpoint ← ⌊(high + low)/2⌋
    res ← TEMP-ALLOC(midpoint)
    if SUCCESS(res) then
        return DEDUCE(midpoint, high)
    else
        return DEDUCE(low, midpoint − 1)
```

Hole	Total	Max
A	0B	131068GiB
B	0B	1028GiB
C	0B	4GiB

Table 3: Initial state for a PIE executable

B PAP-only algorithm

For each hole G, we maintain two variables, G_{total} and G_{max}. The first variable tracks the number of bytes allocated from G. The second holds the maximum number of bytes that may still be allocatable from G at any point in time. So when an allocation of size S is known to originate from G, we increase G_{total} by S and decrease G_{max} by S. Crucially, when an allocation of size S fails, we know that no hole has S bytes available. Therefore, the *max* variable for every tracked hole needs to be adjusted to S minus the pagesize (see the functions HOLE-SATISFIED, HOLE-FAILED-TO-SATISFY, called for holes that satisfied or failed to satisfy an allocation request, respectively).

```
function HOLE-SATISFIED(size, G)
    G_max ← G_max − size
    G_total ← G_total + size
function HOLE-FAILED-TO-SATISFY(size, G)
    if G_max > size − pagesize then
        G_max ← size − pagesize
```

A state consists of the set of *max* and *total* variables for each tracked hole. In the initial state we may have some information for the maximum size of each hole, but no bytes have been allocated during the run of our algorithm, so that $G_{total} = 0, \forall G$. Given the size distributions in Table 1, the initial state for a simple PIE executable is as given on Table 3.

Descent mode The algorithm operates in two modes. Suppose that the highest maximum value (*hmv*) is unique across all the tracked holes and G is the hole it is associated with (i.e. $\nexists H : G_{max} = H_{max}$). In this case, we try decreasing allocation sizes which can only be satisfied by G, in the hope that an allocation will succeed, causing G_{max} to be decremented below the next-highest maximum (*nhmv*) value so that we will remain in this mode for the next step of the algorithm.

There is an inherent tradeoff between the accuracy and the number of allocations we try. In the extreme, we could explore the interval $[nhmv, hmv]$ by starting with *hmv* and decreasing the allocation size by one page after each failed attempt. Of course, this would result in a huge number of allocation attempts, rendering the approach impractical.

```
function CALCULATE-STEPS(high, low)
    size ← high − low
    if size = pagesize then
        return [high]
    step ← size/split
    sizes ← []
    idx ← 0
    for n in 0..(split − 1) do
        sz ← high − n ∗ step
        rem ← sz % pagesize
        if rem > 0 then
            sz ← sz − rem + pagesize
        if idx = 0 ∨ sizes[idx − 1] ≠ sz then
            sizes[idx] ← sz
            idx ← idx + 1
```

While larger successful allocations are desirable, we elect to trade some resolution for a reduction in the number of attempts necessary. The way we do this is by selecting a *split* factor for the interval $[nhmv, hmv]$, and trying decreasing allocations with an (approximate) step of $\frac{hmv-nhmv}{split}$ bytes (special considerations need to be made for respecting page boundaries; see the CALCULATE-STEPS function). A larger split factor results in more allocation attempts but higher chanches of quickly minimizing the *max* variable of G.

At every given step, the allocation might succeed (in which case we update G_{total} and G_{max}) or it might fail and we appropriately reduce every *max* variable to just below the failed allocation size. This means that the difference between the current *hmv* and the *nhmv* keeps shrinking as allocations fail.

For reasons that will become apparent soon, we are willing to expend more allocation attempts to avoid the situation when *hmv* becomes equal to the *nhmv*. Therefore, when the last step above the *nhmv* results in a failed allocation, we reiterate the algorithm, again splitting the interval between the current *hmv* and the *nhmv* according to the split factor and trying descending allocation size using a new, smaller, step size. The algorithm continues trying ever smaller allocations using an ever finer step size, until the allocation of $nhmv + pagesize$ bytes. If that allocation fails, then we have to switch into the mode where there are multiple highest maximum values (Algorithm 2, line 34).

Forking mode We are now in a state where there exist n holes, $G^1, ..., G^n : G^1_{max} = ... = G^n_{max} = hmv$, i.e. the *hmv* has multiplicity n. The only way to make progress is to try an allocation smaller than *hmv*; yet if the allocation succeeds, we are not in a position to tell which hole the bytes where allocated from. What's worse, more than one hole might be able to accommodate the allocation we attempt.

Algorithm 2 Decision

```
 1: function STATE-MAXES(s)
 2:     res ← ∅
 3:     for all max ∈ MAXES(s) do
 4:         res ← res ∪ max
 5:     sorted_maxes ← SORT-DESCENDING(maxes)
 6:     groups ← GROUP-BY-MAX(sorted_maxes)
 7:     result ← ∅
 8:     for all g ∈ groups do
 9:         result ← result ∪ (ANY-MAX(g), COUNT(g))
        return result
10: function DETERMINE-GROUPS(states)
11:     maxes ← ∅
12:     for all s ∈ states do
13:         maxes ← maxes ∪ STATE-MAXES(s)
14:     maxes ← SORT-DESCENDING(maxes)
15:     groups ← GROUP-BY-MAX(maxes)
16:     result ← ∅
17:     for all g ∈ groups do
18:         g ← SORT-DESCENDING-BY-MULTIPLICITY(g)
19:         maxval ← FIRST(g)
20:         result ← result ∪ maxval
21:     return result
22: function DECIDE(states)
23:     maxvals ← DETERMINE-GROUPS(states)
24:     (hmv, m) ← FIRST(maxvals)
25:     if hmv <= mshs then
26:         return states
27:     if COUNT(maxvals) = 1 then
28:         nhmv ← mshs + pagesize
29:         if hmv = nhmv then
30:             return states
31:         sizes ← CALCULATE-STEPS(hmv, nhmv)
32:         states ← DESCEND(states, m, sizes)
33:         return states
34:     (nhmv, ) ← SECOND(states)
35:     if nhmv <= mshs then
36:         nhmv ← mshs + pagesize
37:     if hmv = nhmv then
38:         return states
39:     sizes ← CALCULATE-STEPS(hmv, nhmv)
40:     return DESCEND(states, m, sizes)
```

This constitutes a second mode of operation for our algorithm. After we select an allocation size T, we attempt the to allocate T bytes n times. If all n allocations succeed, the max values for the n holes all get reduced by the same amount; if the new maximum values are still higher than any other maximum value, we remain in the same mode. If there is now a unique (necessarily different) *hmv*, we switch modes.

We then consider the case when k out of n allocation attempts succeed. The allocations could have come out of any set of k holes and there are $\binom{n}{k}$ possible ways to pick the successful holes out of the n we started with. At this point, we fork the current state into $\binom{n}{k}$ new ones, one for each possible combination. In each newly-created state, k holes get their $total$ variables incremented and their *max* variables decremented by the allocated size, whereas the maximum values of the remaining $n-k$ states are lowered to below the allocated size (as we consider the allocation from those holes to have been a failure). After a fork, we are left with a number of active states, one of which matches the actual system state as regards the total allocated bytes for each hole.

Finally, when all n allocations fail, we need to pick a new, smaller allocation size. The obvious approach would be to halve the size for the next allocation attempt. However, this choice leads to a pathological situation. Recall that every failed allocation attempt causes all maximum variables to be set to one page below the attempted value so that, as long as the allocations all fail, the maximum values are reduced in lockstep. Consider the case when an allocation succeeds; the new maximum value for a hole G which is considered successful is set to $G'_{max} = G_{max} - allocation_size = G_{max} - G_{max}/2 = G_{max}/2$. Regarding a hole H for which the allocation was considered a failure, the *max* variable is updated to $H'_{max} = allocation_size - pagesize = H_{max}/2 - pagesize$. Since G_{max} was equal to H_{max}, the new maximum values for all the n holes will all be within a page of each other, which makes it all but certain that after one descent step we will re-enter the forking mode, which increases the chances of a combinatorial explosion in the number of active states.

To increase our chances switching back to descent mode, we elect to pick the next allocation size exactly as we do when the *hmv* is unique.

Generalized Algorithm We extend the algorithm described above to operate when there is more than one active state. Our driving concern is to be able to differentiate between active states. To that end, we collect the highest maximum value of each state. We then attempt a descent from the highest maximum value to the next-highest maximum value using the split factor, as described for the single-state case.

Algorithm 3 Descent

```
function ALLOCATE(m)
    count ← 0
    for i in 0..m do
        if ALLOC( )then
            count ← count + 1
    return count
function NSTATE(size, rest, satisfied, not_satisfied)
    holes ← rest
    for all G in satisfied do
        holes ← holes ∪ HOLE-SATISFIED(size, G)
    for all G in not_satisfied do
        holes ← holes ∪ HOLE-NOT-SATISFIED(size, G)
    return CREATE-STATE(holes)
function DO-COMBINE(rest, selected, previous, count, candidates, accum)
    if count = 0 then
        nstate ← NSTATE(size, rest, selected, previous)
        accum ← accum ∪ nstate return accum
    else if EMPTY(candidates) then return accum
    else
        x ← FIRST(candidates)
        candidates ← TAIL(candidates)
        accum ← DO-COMBINE(rest, selected ∪ x, previous, count − 1, candidates, accum)) return DO-COMBINE(rest, selected, previous ∪ x, count, candidates, accum)
function COMBINE(rest, count, candidates)
    DO-COMBINE(rest, ∅, ∅, count, candidates, ∅)
function UPDATE-STATES(states, size, count, m)
    nstates ← ∅
    for all s ∈ states do
        candidates ← ∅
        rest ← ∅
        for all G ∈ HOLES(state) do
            if G_max ≥ size then
                candidates ← candidates ∪ G
            else
                rest ← rest ∪ G
        if COUNT(candidates) ≥ count then
            res ← COMBINE(rest, count, candidates)
            nstates ← nstates ∪ res
    return nstates
function DESCEND(states, m, sizes)
    for size in sizes do
        count ← ALLOCATE()
        states ← UPDATE-STATES(states, size, count, m)
        if count > 0 then return DECIDE(states)
```

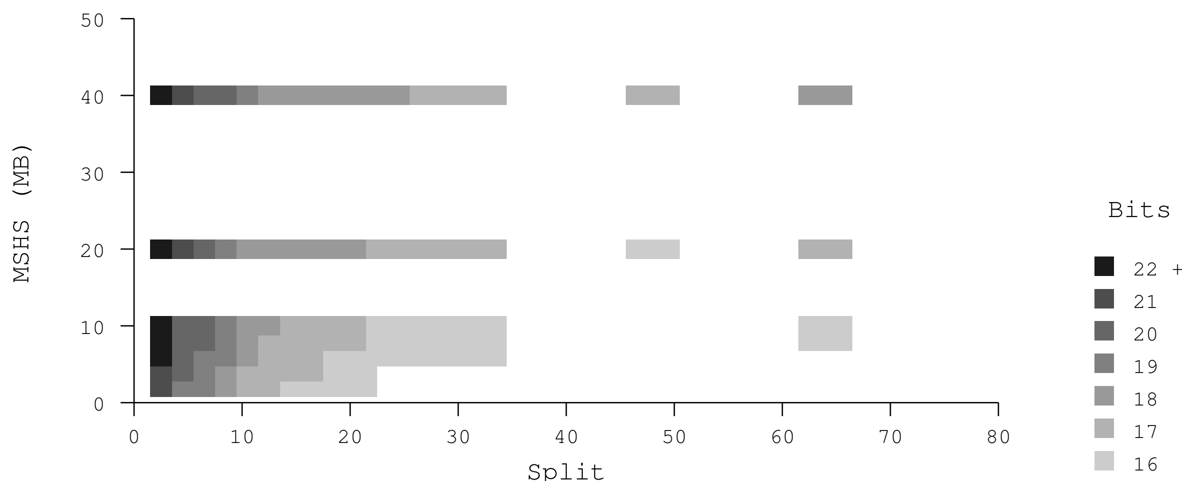

Figure 4: Residual entropy for different value combinations of the split and MSHS parameters (lighter is better)

A crucial insight is that when an allocation succeeds, all states that have a highest maximum value which cannot accommodate the successful allocation size are necessarily impossible and are pruned from the set of active states. This serves to contain the number of active states and somewhat ameliorate the combinatorial behavior of the forking mode.

The generalized algorithm (Algorithm 2) treats all cases (single or multiple states, the *hmv* is unique or has multiplicity n) uniformly. First, it considers the maximum values of the holes within a single state. Multiplicity is established within each state. Following that, the holes are sorted in descending order based on their maximum values and then again in descending order with regard to the multiplicity of each maximum value in the state it originated from. The number of times an allocation is repeated is determined by the multiplicity of the topmost hole; the descent takes place between that hole and the next hole of a different maximum value.

Intuitively, if we have exactly two states, S_1, S_2 with unique *hmv* values hmv_1, hmv_2 and that $hmv_1 = hmv_2$. Then we need try an allocation which can only be satisfied by the maximally-sized hole of either state only once; if the allocation succeeds, both states are updated and remain valid. Conversely, if hmv_1 has multiplicity two and hmv_2 only one, we need to try the allocation two times. If both allocations succeed, S_1 gets updated accordingly and S_2 gets dropped as invalid; if only one allocation succeeds, S_1 is replaced by two new states whereas S_2 is adjusted and remains live.

C Further evaluation of the PAP-only attack

When considering Figure 4 visualizes the residual accuracy (in bits) in the size of the largest hole next to the hidden object. The corresponding figure for the smaller hole appears almost identical and is omitted for brevity.

For the split factor, we investigated values ranging from 4 to 64, specifically 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 48 and 64. Guided by a sampling of typical applications on workstations and servers, we considered the following upper bounds for the sizes of untracked holes: 2MiB, 4MiB, 6MiB, 8MiB, 20MiB and 40MiB.

Our evaluation of the PAP in weakening ASLR's protection of a hidden object involved seeding the algorithm with an initial state consisting of 4 hole descriptions. Specifically, we included the maximum possible values for the holes resulting from the placement of the hidden object at a random address within hole A. Hence, our tracked holes were Large, Small, B and C.

Analyzing Figure 4, we notice the tendency for larger split values to result in lower uncertainty. This tendency is not consistent and may vary with the MSHS; for example, split=48 outperforms split=64 for larger MSHS and is overall the best choice at larger MSHS values.

The split factor of 4 is by far the worst choice. It should be mentioned that the runtime performance deteriorates to the point of being impractical when using 2 as the split factor (for reasons expounded on in B, which led us to exclude it from the parameter exploration.