The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations

Lorenzo Binosi*
lorenzo.binosi@polimi.it
Politecnico di Milano
Milan, Italy

Gregorio Barzasi*
gregorio.barzasi@mail.polimi.it
Politecnico di Milano
Milan, Italy

Michele Carminati
michele.carminati@polimi.it
Politecnico di Milano
Milan, Italy

Stefano Zanero
stefano.zanero@polimi.it
Politecnico di Milano
Milan, Italy

Mario Polino
mario.polino@polimi.it
Politecnico di Milano
Milan, Italy

ABSTRACT

Address Space Layout Randomization (ASLR) is a crucial defense mechanism employed by modern operating systems to mitigate exploitation by randomizing processes' memory layouts. However, the stark reality is that real-world implementations of ASLR are imperfect and subject to weaknesses that attackers can exploit. This work evaluates the effectiveness of ASLR on major desktop platforms, including Linux, MacOS, and Windows, by examining the variability in the placement of memory objects across various processes, threads, and system restarts. In particular, we collect samples of memory object locations, conduct statistical analyses to measure the randomness of these placements and examine the memory layout to find any patterns among objects that could decrease this randomness. The results show that while some systems, like Linux distributions, provide robust randomization, others, like Windows and MacOS, often fail to adequately randomize key areas like executable code and libraries. Moreover, we find a significant entropy reduction in the entropy of libraries after the Linux 5.18 version and identify correlation paths that an attacker could leverage to reduce exploitation complexity significantly. Ultimately, we rank the identified weaknesses based on severity and validate our entropy estimates with a proof-of-concept attack. In brief, this paper provides the first comprehensive evaluation of ASLR effectiveness across different operating systems and highlights opportunities for Operating System (OS) vendors to strengthen ASLR implementations.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Software and application security**.

*Both authors contributed equally to this research.

KEYWORDS

Address Space Layout Randomization (ASLR), Memory Exploitation Mitigations, Operating Systems Security

ACM Reference Format:
Lorenzo Binosi, Gregorio Barzasi, Michele Carminati, Stefano Zanero, and Mario Polino. 2024. The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3658644.3690239

1 INTRODUCTION

In the context of software exploitation, the more information an attacker can gather about the targeted system, the easier it is to exploit it. In particular, one crucial piece of information for the success of *memory errors attacks* (e.g., buffer overflows) is the exact address of objects inside the memory space of the running program; this information allows an attacker to successfully exploit the software vulnerabilities hijacking the execution flow of the program or leaking sensitive data. Even if these vulnerabilities are well known, they still account for more than 20% of the CVE reported [?].

Address Space Layout Randomization (ASLR) is a security measure developed to improve resilience against exploitation techniques that depend on precise memory locations. The base mechanism of ASLR is the randomization of the memory layout of processes to make exploitation a *game of chance*. Thus, its effectiveness increases as the number of attempts needed to hit a precise object in memory increases. Ideally, the entropy of the memory layout should be high enough to make the chances of a successful attack negligible; we see that this is not always the case in real-world implementations. Moreover, we would like ASLR to randomize every object that is allocated inside the memory of a program at the moment of allocation and with high entropy (so, low predictability on the object position); unfortunately, there is a gap between theoretical capabilities and real-world implementation: as shown in this work, on many systems, ASLR lacks at least one of the mentioned aspects. In particular, the memory is often grouped in sections or groups of sections and then randomized all together at the program launch; as a consequence, even if the entropy of single memory objects is high, it can be reduced by collecting information about other memory objects, as their memory sections of origin may be correlated [?]. Even though the correlation is an already known vulnerability [?], current research still lacks a comprehensive evaluation regarding the severity of this vulnerability in terms of entropy reduction. Moreover, the scope of these studies is often limited, focusing on specific sections and predominantly considering operating systems tailored towards Linux enterprise solutions [?]. Additionally, the recent shift to ARM architecture in Apple devices has not been sufficiently addressed, leaving the effectiveness of ASLR on such systems unknown. In this work, we directly address these gaps.

In summary, our contributions are the following:

- We conduct an empirical evaluation of the ASLR mechanism's effectiveness across various operating systems, including Linux, MacOS, and Windows, focusing on allocation probability and the impact of this probability on the correlation between memory objects.
- We identify instances where the absolute entropy of relevant memory objects is considerably low, which makes bruteforce attacks feasible. Moreover, even when the absolute entropy is high, we identified high correlation paths that can be exploited to reduce guessing space and speed up bruteforce exploits.
- We identify memory objects with non-uniform distributions, meaning that certain addresses are more likely to be chosen for allocating a memory object, consequently reducing its overall entropy.
- We identify a sudden reduction in randomization entropy of shared libraries in Linux systems since the 5.18 kernel release due to the introduction of Linux Folios performance optimization.
- We demonstrate through empirical proofs that current randomness in ASLR implementations can be easily bypassed in a short amount of time, even on the latest versions of the analyzed operating systems. Consequently, we discuss possible mitigation strategies to improve the effectiveness of ASLR.
- We release the code used to sample all memory objects in the OSes under study, as well as the code for our ASLR analysis. Additionally, we provide the dataset employed in our analysis, which can be used to replicate our results or conduct further research. All the materials can be found at: https://zenodo.org/doi/10.5281/zenodo.12786166.

2 BACKGROUND AND MOTIVATION

Analyzing ASLR performance is crucial, as ASLR represents one of the final security defenses that need to be overcome for exploiting a system. It is worth noting that ASLR is not the sole defense mechanism against attacks; indeed, additional protections such as the *NX* bit and *Stack Smashing Protection (SSP)* are also in place to enhance security in case a new vulnerability emerges. However, the fact that we can rely on other security mitigations does not excuse vendors from implementing a robust ASLR mechanism. Moreover, being aware of the specific OSes' limitations is particularly useful when choosing which best fits a particular scenario's security requirements and threat model.

In the subsequent sections, we examine the most common ASLR weaknesses, justify the selection of the specific operating system under analysis in this work, and provide a threat model where ASLR should mitigate possible attacks.

2.1 Evaluate ASLR Performances

To understand the limitations of ASLR implementation, we start by defining its goals and expected performance: to increase the average number of attempts an attacker must make to correctly guess a memory address through the obfuscation of the memory layout. In other words, it is a mechanism that frequently randomizes memory objects with low predictability and high granularity. More formally, we utilize the taxonomy proposed by Marco-Gisbert and Ripoll [?], which classifies the capabilities of ASLR implementations into three categories: when, what, and how. By examining the performance in these categories, we assess the effectiveness of an ASLR implementation and identify areas for improvement.

When. The differentiation in ASLR mechanisms across various OSes lies when they perform randomization, i.e., in the frequency with which they refresh the randomization of their memory sections. Ideally, a system would randomize every memory object at the moment of allocation; however, this is challenging due to performance degradation and implementation complexity. Consequently, no commercial OS currently adopts such an advanced technique. The most practical goal is to have the memory randomized every time a new process is allocated in memory (as in the case of Linux). In contrast, some OSes like MacOS [?] and Windows [? ?] only randomize parts of memory and only at boot. However, this approach is considered a flawed implementation of ASLR, as a local attacker could use information gathered from other processes to locate the library position precisely; moreover, a remote attacker could bruteforce the address starting from the lowest possible value, to the highest one, or perform a byte-per-byte attack [?].

What. Another critical aspect concerns the granularity of what is randomized and the number of sections randomized: even a single non-randomized object can be leveraged to take over a system. For instance, certain implementations fail to randomize all executable objects or do so only at boot-time, significantly reducing the system's overall security and increasing the likelihood of successful Return-to-Library (Ret2Lib) or Return-Oriented Programming (ROP) attacks.

How. The last category concerns how objects are randomized, specifically in terms of the number of bits randomized and the relative positioning of the objects. Current implementations of ASLR primarily adopt the *Partial-VM* randomization strategy. This method partitions virtual memory into segments where objects are placed. For instance, on a 64-bit Windows 11 OS, libraries and the executable are allocated in a specific region ranging from the address `0x7ff800000000` to `0x800000000000`; no other object can be allocated in such a region. Although this approach significantly simplifies the ASLR implementation, it also increases the likelihood of generating strong correlation paths.

2.2 ASLR Implementation Weaknesses

All aspects mentioned in Section 2.1 impact the unpredictability of memory layout, which can be quantified using entropy. In particular, we refer to Absolute Entropy as the number of bits that represent the randomization of a memory object. If the probability of allocating an object follows a uniform distribution, then the Absolute Entropy would be the base-2 logarithm of the total number of distinct positions a memory object can occupy within the virtual address space. Nonetheless, given that real-world implementations might not follow a uniform distribution, we consider the more general case following the Shannon Entropy formula: $H = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$, where n is the total number of possible distinct positions the memory object can occupy, and $p(x_i)$ is the probability of the memory object occupying the i-th position. We consider *bits* as our reference measure since, under a uniform distribution, the entropy value is the number of bits altered in an address by ASLR. Instead, we refer to Correlation Entropy as the number of bits representing the randomization between two memory objects, for which we also utilize the Shannon Entropy formula. Here, n denotes the total number of potential differences between two memory objects in terms of memory addresses.

The lack of Absolute Entropy and Correlation Entropy is the primary weakness of ASRL implementations.

2.2.1 *Low Absolute Entropy.*

The major problem affecting ASLR implementations is low Absolute Entropy. In this context, low Absolute Entropy is directly associated with the amount of bruteforce effort an attacker needs to perform to correctly guess the position of a memory object without using any particular technique to predict the position.

In modern OSes, the MSb divides the user space virtual memory (MSb = 0) from the kernel space virtual memory (MSb = 1), and thus, only 47 bits are available for addressing memory in user space. Therefore, due to the page offset, the maximum entropy achievable in a 64-bit system is $47bits - 12bits = 35bits$ for an Intel x86_64 system and $47bits - 14bits = 33bits$ for an ARM M1 system. Unfortunately, as empirically shown in this paper, no 64-bit OS reaches these entropy measures, as additional factors influence its estimation. Firstly, certain sections, such as the heap and stack, require room to expand, limiting the potential for placing such objects in memory since they necessitate space to grow upwards or downwards. Secondly, non-contiguous sections like the heap and libraries must have large memory ranges available for random placement. Thirdly, flawed RNG mechanisms can also contribute to reduced entropy.

2.2.2 *Low Correlation Entropy.*

The correlation between various memory objects can significantly impact the complexity of an attack, as the leak of one memory address may significantly reduce the effort needed to compromise another. This reduction in complexity can be quantified by assessing Correlation Entropy, which measures the entropy of the offset between two memory objects. We distinguish two scenarios:

Correlated Objects. When the addresses of allocated memory objects are correlated, their Correlation Entropy is lower than the Absolute Entropy of either object individually. Consequently, a vulnerability that leaks the address of one object can be exploited to de-randomize other objects. In other words, the knowledge of the address of one object reduces the search space for the address of the other one.

Independent Objects. Two randomly allocated memory objects are independent when the Correlation Entropy between these objects is higher or equal than the maximum Absolute Entropy of the two objects. Hence, it is easier to guess the absolute position of an object directly.

Not all correlation scenarios are easily exploitable. In fact, if the Correlation Entropy is sufficiently high, we can consider the object to be secure, even if it exhibits a low correlation with other objects. The problem arises when the correlation is so high that an address leak could enable bruteforcing the position of other objects within a reasonable timeframe, which, in our analysis, was considered to have 20 bits of entropy, as we explain in Section 5. When this occurs, we can identify a *Correlation Path* that could potentially be exploited. The most severe case of correlation presents 0 Correlation Entropy because the offset is fixed. This issue was exploited in the famous `off2libc` attack [?] and led to the introduction of the Effective Entropy concept into ASLR-related discussion [?].

2.3 ASLR Hardware Limitations

The underlying hardware plays an important role in the randomization of memory objects. Theoretically, the maximum entropy for an object is determined by the address size, which is 64 bits or 32 bits according to the hardware architecture. However, the practical limit is lower because of *Multilevel Paging* and *memory pages.*

Multilevel Paging is a technique that translates virtual memory addresses into physical ones. The idea is to split the address into several parts, each serving as an index into a different level of the page table hierarchy. For instance, in Intel's 4-level paging, an address is divided into four segments of 9 bits each, plus a 12-bit page offset (12 *Least Significant Bits* (LSb)), for a total of 48 bits. Consequently, the 16 *Most Significant Bits* (MSb) are not supported at the hardware level – they are represented as the sign extension of the address – which leaves only 48 bits available for addressing memory. Moreover, even with the latest 5-level paging implementation, the practical limit for ASLR remains at 56 bits.

On the other hand, memory pages are the smallest unit of memory that a kernel can allocate, and they are always allocated aligned with the page's size. Therefore, the LSb of an address represent the offset within the page, and they cannot be used in the randomization process. For instance, in Intel x86_64 systems we have 4KB pages, and thus the 12 LSb are used for the page offset, while in ARM M1 systems (64 bits) we have 16KB pages [?], and thus the 14 LSb are used for the page offset. Hence, the bits that can be used for randomization are $48bits - 12bits = 36bits$ for Intel x86_64 systems and $48bits - 14bits = 34bits$ for ARM M1 systems.

In 32-bit systems, Multilevel Paging covers all of the bits of an address, but we still have 4KB pages. Therefore, the maximum entropy for an object is $32bits - 12bits = 20bits$, which is relatively low, making ASLR less effective in 32-bit systems. For this reason, we exclude 32-bit systems from our analysis.

2.4 ASLR Base Addresses

Other factors that can weaken ASLR effectiveness are *ASLR base addresses.* An ASLR base address is an address, chosen either at runtime or boot-time, used to allocate a sequence of memory objects, either after or before the base address. These objects are allocated by applying an offset to the previously allocated object, and the offset can be one of the following: *zero*, *fixed*, *alignment*, or *random*.

When the offset is zero, objects are allocated contiguously. This is common in most OSes when allocating executables, where all sections (e.g., `.text` and `.bss`) are placed contiguously to maintain cross-references and improve performance. A fixed offset, typically of at least one page, is often used for guard pages, like those in the Scudo [?] Android allocator, to prevent buffer overflows. An alignment offset is used to align pages of different sizes, such as aligning a huge page to its size. Finally, a random offset, typically multiple of a page's size, is used to allocate an object at a random position from the previous one and is the only offset that can increase the Correlation Entropy between objects. However, if the random offset is too small, the Correlation Entropy remains low, allowing attackers to reduce the search space for correlated objects.

For instance, in the latest Linux distributions, we have three main ASLR base addresses: the executable base address, the shared libraries base address, and the stack base address. The executable base address is used for both the executable and the heap. After the executable is allocated, the kernel applies a random offset to its end address to compute the heap address, which is retrieved by the application via the `brk(NULL)` system call. The shared libraries base address handles shared libraries and dynamic memory pages allocated through `malloc()` (when the size exceeds `M_MMAP_THRESHOLD`) or `mmap()`. Here, memory pages are usually allocated contiguously, or with small random offsets, towards lower addresses, resulting in low Correlation Entropy that can potentially lead to Ret2Libc or Ret2LD attacks [?]. Finally, the stack base address is used to allocate the stack, arguments, and environment variables. At program startup, the kernel allocates environment variables and arguments at the stack's top, and afterward, the userspace application allocates functions' frames. Similar to the shared libraries base address, allocations are contiguous with a random offset between environment variables and arguments, once again resulting in low Correlation Entropy.

Although identifying ASLR base addresses may be challenging due to missing documentation or code availability (e.g., in Windows), similar situations occur in other OSes. Generally, the stack base address is present across all OSes, while others could be different as they may allocate different objects. Despite potentially weakening ASLR effectiveness, base addresses are a common practice across OSes to improve performance and simplify ASLR implementations. In Section 5, we will see that most correlated objects are due to the ASLR base addresses.

2.5 OSes Choice Motivation

Looking at research related to ASLR analysis, we can see a stable trend toward Linux systems. This is justified by the predominance of Unix servers active in 2023, counting for around 80% of the market share, of which around 50% uses Linux kernels [?]. Enterprise servers require higher security standards than consumer systems, so, understandably, those have received more attention, sometimes evaluating even hardened versions of the Linux kernel available on the market. On the other hand, we have the consumer market, where Linux-based systems count for 3% of the market share when compared with 70% of Windows and 20% of MacOS systems [?]. Moreover, the recent adoption of ARM architectures by Apple made all the research regarding MacOS outdated since they focus only on x86_64 architectures and, as discussed in Section 2.3, the virtual address translation is handled differently [? ?]. Therefore, we decided to focus our research on the following operating systems: Windows (version 11), macOS (Ventura 13.4.1), and Linux (Ubuntu 22.04). We excluded Android from our research, despite its Linux kernel base, as we focused solely on desktop systems. Moreover, Android presents inherent challenges that require more in-depth analysis, such as variability in memory page sizes, differences across devices due to vendor-specific customizations, the non-deterministic Scudo heap allocator, and the difficulty in accessing memory objects managed by the Android Runtime (ART) environment. For a more detailed explanation, we refer the reader to Appendix A of the extended version of this paper [?].

Finally, as discussed in Section 2.3, hardware architecture plays a crucial role in the randomization of memory objects. Thus, we consider the x86_64 architecture for Windows and Linux, and the ARM M1 architecture for macOS. For Linux, we consider two kernel versions (5.17.15 and 6.4.9) to study the impact of Memory Folios on Linux ASLR implementation. For macOS, we consider both the native ARM M1 architecture and the Rosetta dynamic binary translator, which allows applications compiled for x86_64 to run on Apple ARM processors like the M1 chip.

2.6 Threat Model

We consider two exploitation scenarios: ① *Local Exploitation* and ② *Remote Exploitation*. In the first scenario, the attacker has access to the target system, either remotely or physically, and can run arbitrary code. The attacker thus knows the OS, and the goal is to leak sensitive data or to perform a *privilege escalation*, i.e., to obtain superuser privileges. In the second scenario, the attacker does not have access to the target system but can interact with it through an exposed service. In this scenario, the attacker aims to achieve one or more of the following goals: *Remote Code Execution (RCE)*, *Privilege Escalation*, or *Data Leakage*. In addition, the attacker does not know the OS nor its version and must gather information about the system to exploit it or try exploits for different OSes and versions.

In both scenarios, the attacker aims to exploit a memory corruption vulnerability, such as a buffer overflow or a use-after-free, on a userspace application to achieve the aforementioned goals. Hence, depending on the application and the vulnerability, the attacker may corrupt data structures, function pointers, and flow-related variables in the virtual memory of the target process. In general, the attacker needs to know the position of relevant memory objects in order to perform the exploitation. For instance, whenever the attacker can change the return address of a function, performing a *Return-Oriented Programming (ROP)* attack, the attacker needs to know the position of the gadgets in the memory. These gadgets can be found in the executable and in shared libraries. Therefore, the attacker either has a *memory leak* vulnerability that reveals the positions of these objects or needs to guess their positions. As we will see in Section 5 and Section 6, the attacker may reduce the guessing space with a memory leak of another correlated object. It is important to note that the relevance of a memory object depends on the specific vulnerability and the target application. Most of the time, the attacker is interested in the executable and the libraries, as they contain code to perform control flow hijacking. However, in some cases, the attacker may be interested in other objects, like the heap, to perform a *Heap Spraying* attack, or the stack, to target flow-related variables and hijack the flow of the application.

When the exploit fails because the object is not in the guessed position, we usually have a *Segmentation Fault (SEGFAULT)* signal, and the process is terminated. In the local scenario, the userspace application can be restarted, and the attacker can try the exploit again. In the remote scenario, the exposed services usually have a parent process that waits for incoming connections and delegates the connection to a child process. Therefore, in case of a segmentation fault, only the child process is terminated, and the attacker can try the exploit again by opening another connection.

Finally, we assume an attacker capable of performing 300 *tries per second (tps)*, i.e., the number of exploit attempts an attacker can perform in a second [? ?]. We consider 300 tps as a reference value as, at this rate, an attacker can guess the position of a memory object with an entropy of **20 bits** in approximately 1 hour. This value can be more or less realistic depending on the target application logic, the hardware on which the application runs, and the exploit's size. On consumer-grade hardware, we observed tps ranging from 30 to 500 with one core. In Section 6, we will provide an instance of a real-world application where we achieve approximately 300 tps with an exploit running on a single core and more than 1,000 tps with the same exploit running on multiple cores. Remotely, estimating tps is more challenging due to additional factors like the proximity to the victim machine and the network speed [?]. However, 300 tps remains a realistic estimate even in remote scenarios, as attackers can parallelize the exploit across multiple threads, cores, and machines. Additionally, in a remote scenario, the attacker can perform the exploitation using machine(s) geographically close to the target one, reducing the network latency. As references, with 300 tps, an attacker can guess a memory object with 30 bits of entropy in around 41 days, a memory object with 25 bits of entropy in around 1 day, a memory object with 20 bits of entropy in around 1 hour, and a memory object with 15 bits of entropy in around 2 minutes.

3 RELATED WORKS

The most advanced tool used in research is **ASLR-A** by *Marco-Gisbert and Ripoll* [? ?]. It was used to perform analysis on Linux 4.15, PaX (a hardened version of Linux kernel), and MacOS (originally referred to as OS X). As mentioned before, we will consider only the consumer OSes, so PaX implementation is out of the scope of this research. The tool was developed to overcome the limitation found in *paxtest*, a tool developed by the PaX team to evaluate the performance of their newly developed ASLR implementation. *paxtest* had several issues. It considered only Absolute Entropy, using a custom heuristic not always accurate when dealing with non-uniform distributions. Moreover, the limited number of tests does not provide statistical significance to the results. Finally, the analysis was incorrect in some cases; the sampling of text area was, in reality, the library section.

They improved those aspects by developing ASLR-A, a tool capable of taking thousands of samples at a second and able to analyze numerous statistics. However, in this document, we focused mainly on two aspects that are the ones easily exploitable in bruteforce attacks: **Absolute Entropy** and **Correlation Entropy**. The tool can provide Absolute Entropy estimation using three different methods: *Shannon*, *Shannon* at byte level, *Shannon* with variable bins width, and *bit-flipping*. Based on [?], it seems to be capable of estimating also Correlation Entropy. However, the last known version of the tools available on the researcher's website [?] provided only a correlation matrix without the estimation of Correlation Entropy. In the end, the tool provides a good insight into the Probability Distribution of sections.

The only true limitation we can identify in this research is the limited scope of objects and OSes considered. In fact, as mentioned in the previous section, ASLR performance is related to many runtime conditions, such as memory fragmentation, thread execution, and allocation patterns, so the allocation of multiple objects per section and multiple threads can lead to a change in randomization performance. The same considerations are valid for their MacOS analysis. However, it is not clear how the samples for this system were collected as the randomization is performed only at boot-time. No other research is available on the MacOS platform.

To the best of our knowledge, there are only three published studies on Windows. The first, regarding Windows Vista [?], is outdated, so we will focus only on the ones analyzing Windows 10 [?] and Windows 7 [?]. The sampling of Windows 10 was performed through `5000` reboots using a custom-written tool, which took a total of `500,000` samples, while for the sections that were randomized at runtime `5 mln` samples were considered [?]. The results are not publicly available but, based on the researcher's claims, they were able to estimate the Absolute Entropy of memory objects, probability distribution, and their correlation; however, no mention of Correlation Entropy was made, and they just considered the main execution flow, without launching multiple threads; moreover, as in the case of ASLR-A, no attention to doing multiple allocations of different sizes where taken [?].

The analysis of Windows 7 [?], even if it is outdated and considered only four memory sections, concluded that the problems highlighted in Windows Vista [?] were still present: heap-allocated objects with non-uniform distribution and shared libraries randomized at boot-time.

Over the years, many pointed out that PRNG on Android has low entropy [? ?]. Moreover, because every process is forked from Zygote, we can expect poor runtime performance of ASLR [?]. Another problem of Android security is the customization made by vendors [?]. This aspect is hard to analyze due to the fragmentation of the Android hardware and vendors. Thus, we decided to focus the analysis on desktop OSes only.

3.1 Limitations and Improvements

All mentioned researches have at least one of the following limitations: ① Lack of a broad OSes analysis, ② missing thread execution, ③ inadequate sampling size, ④ limited considered sections, ⑤ limited allocations, ⑥ missing Correlation Entropy estimation, or ⑦ unclear entropy estimator choice.

This last point is strictly related to the inadequate sampling size. For example, to obtain an accurate estimation using direct Shannon entropy, we need $\mathbf{O}\left(\frac{\mathbf{k}}{\log(\mathbf{k})}\right)$ samples where $\mathbf{k}$ is the number of symbols considered [**?**]. As mentioned before, the maximum entropy obtainable on the considered system is 35 bits, therefore a k size of $\mathbf{2^{35}}$. To Estimate the entropy using the Shannon formula, we will need $\frac{2^{35}}{\log_{10}(2^{35})} = \mathbf{3.261.159.434}$ samples which are way too much to be collected in a reasonable time. Even if we consider the best in class, Linux, which in some sections comes close to 30 bits of entropy, we are still considering hundreds of millions of samples to be collected. This problem is even more relevant when we take into consideration reboot times, so we need a less greedy estimator.

The use of Shannon at the byte level or other sorts of plug-in methods is a good approach, significantly reducing the number of samples needed to obtain a good estimation. However, they tend to overestimate the value of entropy due to outliers or due to non-uniform distribution. The bit-flipping and other bit mask estimators are indicators of the changing bits of the address and only give a rough upper bound to the entropy value.

4 HOW TO EVALUATE ASLR IMPLEMENTATIONS

Assessing the effectiveness of ASLR implementations is a challenging task due to the dynamic nature of virtual memory allocations in a program. These allocations can occur at various stages, such as boot-time, program startup, or runtime. Therefore, to understand how these memory allocations are related, it is necessary to conduct an empirical analysis. This is done by collecting the memory addresses of different objects and sections of the program, over multiple runs, and then analyzing the randomness of the memory layout and the correlation between different objects. In this way, through the statistical analysis of millions of samples, we evaluate the strengths and weaknesses of ASLR implementations.

4.1 Sampling

For the sampling phase, the main focus is efficiency and granularity of information. We wanted to emulate the behavior of real-world software to provide information about the effort needed to hit a specific object in memory and not only the page it belongs to. On the one hand, we need as much data as possible to better analyze the ASLR performance across the different operating systems. On the other hand, the resources at our disposal are limited. Fortunately, we can define a unique subset of objects and allocation sizes, shared across all considered platforms, able to provide a solid picture of the ASLR details in an efficient and homogeneous way.

Memory Objects. Facing the problem of choosing which memory object and section to sample in our research, we decided to focus on the interactions and correlations between objects; as a consequence, our sampling program makes multiple allocations of different sizes from 3 different flows: two independent threads (ThA, ThB) and the main thread (M). Because of this, the total number of addresses collected with each sample is around 60 (accounting for some platform limitations). Additionally, for Executable and Linkable Format (ELF) and Portable Executable (PE), we have different sections such as `.text`, `.bss`, and `.data`. In these cases, we do not need to sample all the sections, as they are placed contiguously in the virtual memory of the process. As a result, we only need to sample one section to know exactly the position of all the other sections. Finally, for simplicity, we group all these sections under a single memory object called `executable`.

Table 1: Selected Sizes for `malloc()` allocations.

	16B	512B	4KB	256KB	4MB	128MB
Linux 6.4	`[heap] brk()`			`pages`	`folio pages`	
Linux 5.17	`[heap] brk()`			`pages`		
MacOS M1 13.4.1	`M_NANO`	`M_TINY`	`M_SMALL`	`M_MEDIUM`		`M_LARGE`
Windows 11	`[heap]`				`pages`	

To represent our memory objects, we use the syntax *<object>_-<thread>*, where *object* is the name of the object, and *thread* is the thread to which the object belongs. For instance, *stack_M* is the stack of the main thread. For dinamically allocated object instead, we use the syntax *<function_name>_<size>_<thread>_<sequence_-number>*, where *<function_name>* is the function that allocates the object, *<size>* is the requested size, and *<sequence_number>* is the i-th operation of that type. For instance, *malloc_4KB_ThB_2* is the second 4-kilobytes malloc function call performed by thread B.

Allocation Size Choice. Operating systems often employ various allocation methods for different memory sizes to enhance performance. For instance, Linux uses a default threshold of `128KB`(`M-_MMAP_THRESHOLD`) to decide whether to use the legacy system `brk()` or the `mmap()` function as an allocation method; moreover, this threshold is variable and is optimized at runtime based on the allocation pattern [**?**] of the program so we cannot take that for granted. Since the introduction of Folios in Linux 5.18 [**?**], we have one more variation of allocation method for `mmap()` of sizes `>2MB`, with a significant impact on randomization entropy, as we will see in Section 5.1 As a consequence, we consider two possible allocation methods: the `malloc()` and the `mmap()` functions. We believe they represent the most common allocation methods for our operating systems. In particular, Windows has an equivalent `mmap()` function called `VirtualAlloc()`.

For `malloc()` allocations, we consider different sample sizes ranging from `16B` to `128MB`, as reported in Table 1. In particular, in Windows and Linux, allocations of several kilobytes through the `malloc()` internally result in `mmap()`/`VirtualAlloc()` function calls. As a result, the kernels label these memory areas as general pages instead of heap pages. However, we consider them as heap pages since they are the result of their standard memory allocators.

Instead, for `mmap()`/`VirtualAlloc()` allocations, we consider specific sizes, which are characteristic of the OSes. Specifically, the term Single page refers to `4KB` pages for both Linux and Windows and for `16KB` pages for MacOS. This difference is due to the design implementations; Apple M1 chips are designed to allocate `16KB` to enhance runtime performance. Instead, the terms Huge pages and Large pages refer to `2MB` pages for Linux and Windows, respectively.

4.2 Entropy Estimator

The most important quantitative parameter we want to analyze is Information Entropy, which is a concept introduced by Shannon [**?**] in 1948 that measures the information contained in a data source.

From another point of view, it is a way to measure the randomness. In the context of ASLR, which is a system that incorporates the approach of "security through obscurity", entropy is directly linked to the effort needed to guess the current position of a memory object in terms of trials. Because of that, having a reliable way to estimate the entropy is a crucial part of ASLR analysis.

As mentioned in Section 2.2.1, we are dealing with addresses the size of 47bit (127TB of addressed space), so exhaustively sampling all values of our source is practically impossible. Moreover, to use the Shannon Entropy estimator, we need more than one sample for each bin, so the number is even bigger. We are dealing with an under-sampled discrete source analysis, so we must use an estimator suited for this task. The most common method to estimate Shannon entropy is to consider each byte (or a subset of bytes of the address) as an *independent random variable* and then combine the resulting entropy to estimate the one of the complete addresses (a.k.a. *Plugin Entropy*). Even if, in theory, it is a good method, we considered the assumption about the independence of bytes with regard to each other too strong to be stated generally true. To completely avoid this assumption, we decided to use a not-binned estimator.

The best option we identified is the *Nemenman, Shafee, Bialek (NSB)* estimator [**? ?**], which is a coincidence-based estimator that also provides us with *posterior standard deviation* to quantify the uncertainty in the estimation result. Thanks to this, it is still one of the best estimators for under-sampled sources, outperforming both *Shannon Entropy* and *Plugin Entropy* [**?**]. It has a bias of $\frac{2^{S/2}}{N}$ [**?**] where S is the unknown entropy and N is the number of samples. Thus, we can calculate the number of samples we will need in the worst-case scenario to have a bias of less than 5%.

This method is also suitable for the Correlation Entropy estimation. For correlated objects, we observe that the resultant entropy typically falls below the minimum entropy observed in either object, thereby incurring a bias comparable to or less than that encountered in absolute entropy measurements. Conversely, when evaluating two independent objects, the Correlation Entropy may exceed the individual maximum entropies, necessitating a larger sample size due to the increased bias. Nonetheless, this bias is acceptable since our primary interest is to correctly quantify high Correlation Entropies. Consequently, precise Correlation Entropy values for independent objects are not a priority.

4.3 Sample Collection

Following the rule presented in Section 4.2, we consider the maximum theoretical entropy (i.e., 35 bits) to identify the minimum number of samples. In this case, we want to collect enough samples to have a bias lower than 5%. Hence, $\frac{2^{35/2}}{N} < 0.05$. Solving for N, we find that we need at least 3,800,000 samples.

To collect the required samples, we employ one x86_64 machine and one ARM M1 machine. Additionally, to collect Windows 11 samples, we employ a Virtual Machine (VM) running on Ubuntu 22.04. Such a VM is virtualized to directly access the hardware and avoid any possible bias of the guest OS. However, we have memory objects that are randomized at boot-time, and thus, we need to reboot the system to collect the samples of these memory objects. Due to the significant amount of reboots required and the slow boot-time of the VM, it is not possible to collect the required amount of samples in a reasonable time. Hence, we first identify the *Changing Bitmasks* – i.e., how many bits change in the address of objects in memory over different runs or reboots – to estimate the minimum number of samples to have a bias lower than 5%. It is important to recall that the NSB estimator provides the posterior standard deviation to quantify the uncertainty in the estimation result. Hence, after performing the analysis on the required samples, we evaluate the uncertainty of the estimation to confirm that the bias remains below 5%. In other words, we verify that the estimated *Changing Bitmasks* is therefore correct.

The thresholds for runtime and boot-time randomized objects are reported in Table 2 and Table 3 respectively, where T stands for Theoretical and CB for Changing Bitmask.

Table 2: Thresholds for runtime randomized sections.

OS	Entropy (bits)	Min Samples
Ubuntu	35 (T)	3,800,000
Windows	35 (T)	3,800,000
MacOS	19 (CB)	15,000

Table 3: Thresholds for boot-time randomized sections.

OS	Entropy (bits)	Min Samples
Windows	19 (CB)	15,000
MacOS (M1 Native)	16 (CB)	5,000
MacOS (M1 Rosetta)	15 (CB)	3,600

5 RANDOMNESS ANALYSIS

In this section, we present the results of the analysis as well as the findings on the different OSes with the different configurations. As discussed in Section 2.6, we consider **20 bits** of entropy as the reference threshold. This threshold is artificial as, in reality, there is no real value of "safeness", and it strongly depends on how many tps an attacker can perform. We consider good results to be everything above this value and bad results to be everything under. Moreover, we briefly discuss the strengths and weaknesses of each system considering: ① *Allocation Layout*, ② *Probability Distribution*, ③ *Absolute Entropy*, and ④ *Correlation Entropy*.

To present ① *Allocations Layouts*, we represent the different groups, sections, or objects as horizontal colored bars to qualitatively highlight in which range of addresses each object can be allocated. It is important to note that these layouts just represent possible allocation addresses and not how much they expand during the execution. To present ② *Probability Distributions*, we rely on the *Binned Histogram*, a discrete visualization method that represents the random nature of our memory objects. To present ③ *Absolute Entropies* results, we group objects and sections that are contiguous or with zero entropy to one another. These groups cannot be defined overall as they depend on the OS. For instance, the glibc heap manager allocates all the chunks smaller than a few kilobytes on the same heap page. As a result, we consider all the small allocations to be a single group under Linux. Finally, to present ④ *Correlation Entropies* results, we report the *Distance Entropy Matrices*. These matrices highlight the entropy of the distance between two objects or groups. The reader can find complete Correlation

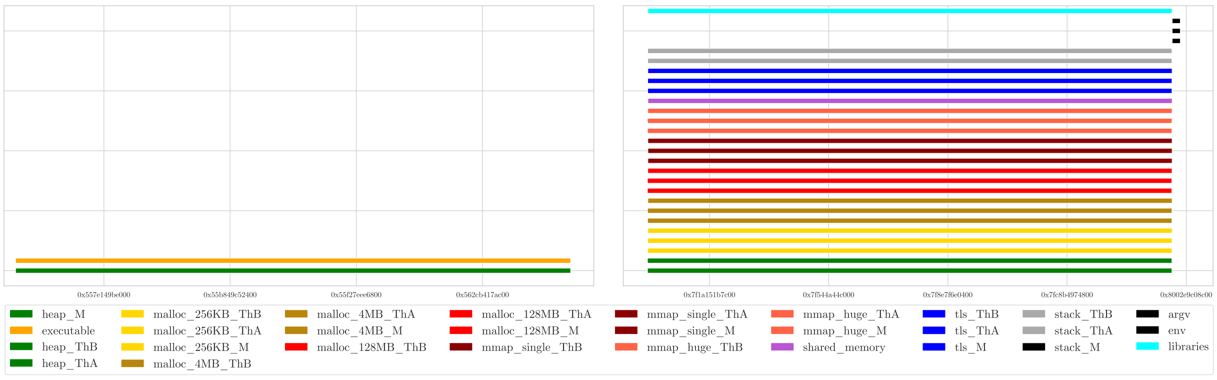

Figure 1: Linux (Ubuntu 22.04) Allocation Layout.

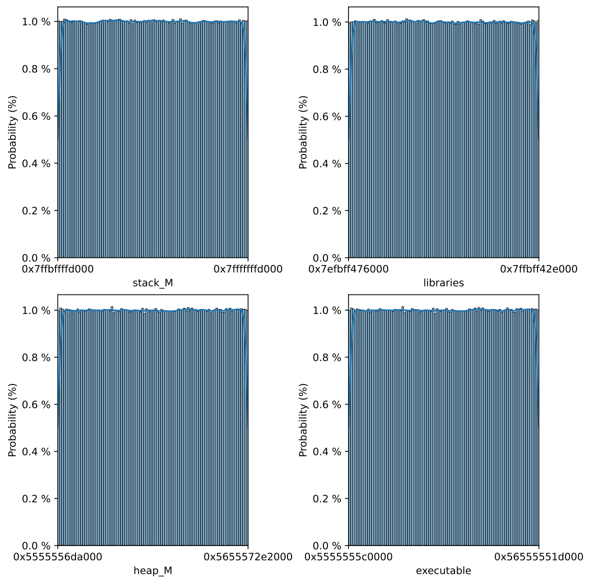

Figure 2: Probability Distribution in Linux (Ubuntu 22.04). All the distributions are available at: https://zenodo.org/records/12968870/files/linux_distribution.pdf

Entropy matrixes in the Appendix B of the extended version of this paper [?]. The complete list of figures is also available at: https://zenodo.org/records/12968870

5.1 Linux

We perform our analysis on Ubuntu 22.04 running on an x86_64 machine, employing the latest glibc library in Ubuntu 22.04, namely glibc 2.35. With the introduction of Folios in Linux kernel 5.18, we analyze two kernel versions: 5.17.15 and 6.4.9. We believe Linux Folios, as well as Huge pages (2MB), can drastically decrease the entropy of memory pages. As a result, we collected more than 4 million samples for each kernel version and evaluated their randomness.

All the samples have been collected without performing any reboot since the Linux kernel randomizes every memory object at runtime. Moreover, the sampling program is compiled with *GCC* as *Position Independent* (flag `-fPIE`) and relies on external libraries, i.e., not compiled with the flag `static`. Finally, the kernels are configured to fully randomize the objects in user space (`/proc/sys/kernel/randomize_va_space` = 2).

Allocation Layout. As we can see in Figure 1, the allocation layout is the same for both kernel versions and is mainly divided into two regions with a large empty region in the middle. On the left, starting with lower addresses, we have the sections of the `executable` and the `heap`. This latter contains all the allocations of the main thread with sizes lower than a dynamic threshold. On the right of the figure, positioned among high memory addresses, we have the `stack`, the `libraries`, and all the allocations that are usually greater than `256KB`, allocated through the `mmap()` or the `malloc()` functions. Additionally, we can find other threads' `stack`, `heap` and `Thread Local Storage (TLS)`. As you can notice, most of the object and group allocations almost completely overlap. This is due to the fact that some objects are allocated considering the allocation of other objects. For instance, the `heap` is allocated after the `exectuable`, considering a random offset starting from the end of the `exectuable`. This behavior is the leading cause of Correlation Entropy since the position of an object depends on another, especially when the random offset is limited. Similarly, the `libraries` are allocated considering a random offset starting from the end of the `stack`. In this case, the offset guarantees the `stack` to grow sufficiently, and this can be seen in the figure since the black bars are separated from the others.

Probability Distribution. Regardless of the kernel version, Linux randomizes every object uniformly. This is evident from the probability distribution in Figure 2. Figure 2 shows a histogram of 100 bins of the same size. The probability of a memory object being allocated in a specific range of addresses is the same for all the objects and is around 1%, as expected for a uniform distribution. Moreover, the allocation range is also reported on the x-axis.

Absolute Entropy. From Table 4 we can see the Absolute Entropy of both kernel versions. The most common and used sections of the main thread, such as the `stack`, the `executable`, the `heap`, and the `libraries`, achieve a relatively secure entropy of 28.8 bits, with the `stack` reaching up to 31.8 bits. However, in the kernel version 6.4.9, we have a drop in the entropy to 19 bits when the library is larger

Table 4: Linux (Ubuntu 22.04) Absolute Entropy. *lib_small* is a library whose size is lower than 2MB. *lib_big* is a library whose size is bigger than 2MB. In our case, *lib_big* is the *glibc*. This distinction is important since Linux Folios are used only for the *lib_big* allocation. In fact, *lib_big* for `Linux 6.4.9` has a lower entropy than *lib_small*.

Object	Entropy		Object	Entropy		Object	Entropy		Object	Entropy	
	5.17.15	6.4.9		5.17.15	6.4.9		5.17.15	6.4.9		5.17.15	6.4.9
env	22.472	22.472	heap_M	28.846	28.850	heap_ThA	14.926	14.978	heap_ThB	14.744	14.931
stack&argv_M	30.832	30.836	malloc_256KB_M__1	28.845	25.832	malloc_256KB_ThA__1	24.247	23.998	malloc_256KB_ThB__1	24.773	24.006
shared_memory	28.845	28.851	malloc_4MB_M__1	28.845	19.611	malloc_4MB_ThA__1	19.752	20.007	malloc_4MB_ThB__1	19.912	20.197
tls_M	28.845	28.836	malloc_128MB_M__1	28.845	19.611	malloc_128MB_ThA__1	16.236	16.071	malloc_128MB_ThB__1	15.660	16.134
lib_big	28.845	19.023	mmap_single_M__1	28.845	28.851	mmap_single_ThA__1	28.838	28.838	mmap_single_ThB__1	28.836	28.838
lib_small	28.845	28.851	mmap_huge_M__1	19.024	19.023	mmap_huge_ThA__1	18.814	18.875	mmap_huge_ThB__1	18.869	18.883
executable	28.862	28.840	malloc_256KB_M__2	25.911	24.902	malloc_256KB_ThA__2	23.722	23.501	malloc_256KB_ThB__2	24.154	23.522
stack_ThA	19.026	19.023	malloc_4MB_M__2	19.588	19.023	malloc_4MB_ThA__2	18.929	19.117	malloc_4MB_ThB__2	18.977	19.125
tls_ThA	19.026	19.023	malloc_128MB_M__2	19.588	19.023	malloc_128MB_ThA__2	16.840	16.786	malloc_128MB_ThB__2	16.227	16.636
stack_ThB	19.026	19.029	mmap_single_M__2	28.845	28.851	mmap_single_ThA__2	28.836	28.843	mmap_single_ThB__2	28.835	28.836
tls_ThB	19.026	19.029	mmap_huge_M__2	19.024	19.023	mmap_huge_ThA__2	18.711	18.804	mmap_huge_ThB__2	18.787	18.808

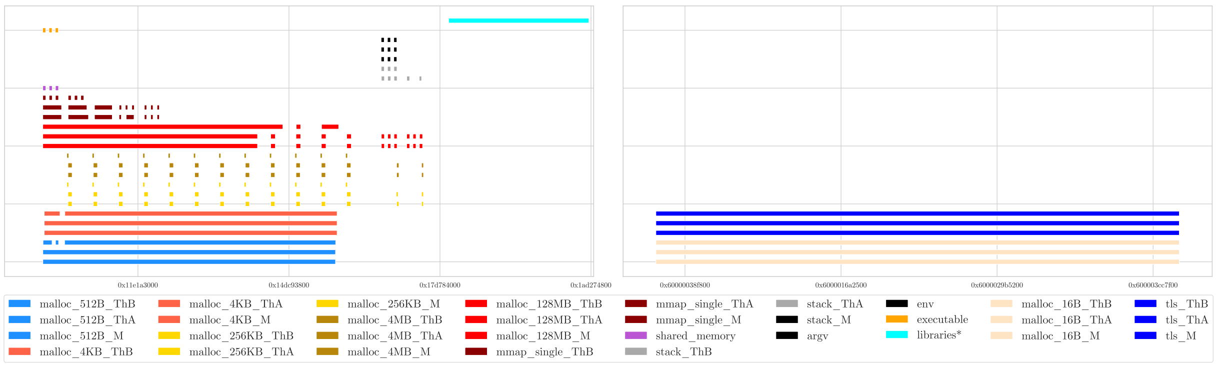

Figure 3: MacOS M1 Native Allocation Layout.

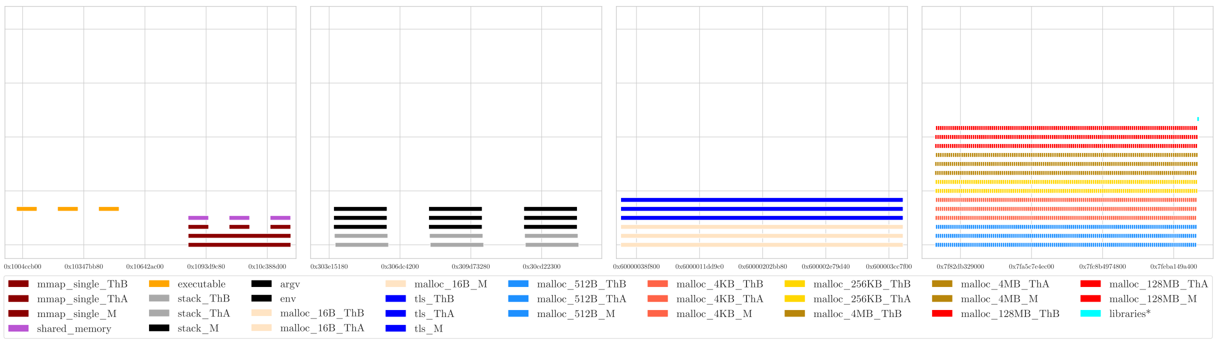

Figure 4: MacOS M1 Rosetta Allocation Layout.

than a huge page (2MB). This is due to the Linux Folio optimization [? ?]. Linux Folio is a memory structure introduced to increase performance and reduce memory fragmentation, as it groups multiple consecutive memory pages of 4KB, as a single bigger memory chunk. This uses neither huge pages nor transparent huge pages and is a flexible structure, so in theory, the size is not fixed. Its size is a power of two, and it is aligned with its size [?], so for a Large Folio of 2MB, we should see a page offset of 21 bits versus the 12 bits of a 4KB page. As a consequence, all memory objects allocated using this new structure should expect around a 9-bit reduction in entropy. In fact, the sampled lib experiencing the reduction is the `glibc` that is indeed bigger than 2MB. This is a huge reduction for an executable memory section and gives an attacker almost four hundred times more chances of success compared to Linux version 5.17.15. Moreover, you can notice an entropy reduction after the *mmap_huge_M__1* in 5.17.15. This allocation indeed reduces the entropy of the next page allocations since they are relative to it; pages are allocated contiguously or almost contiguously and thus, a huge page determines the allocation of the next pages. In 6.4.9, a similar pattern is observed where the first huge page allocated is for *lib_big*, which in this case is `glibc`. We argue that having `glibc` easily guessable poses a serious security issue. Since `glibc`

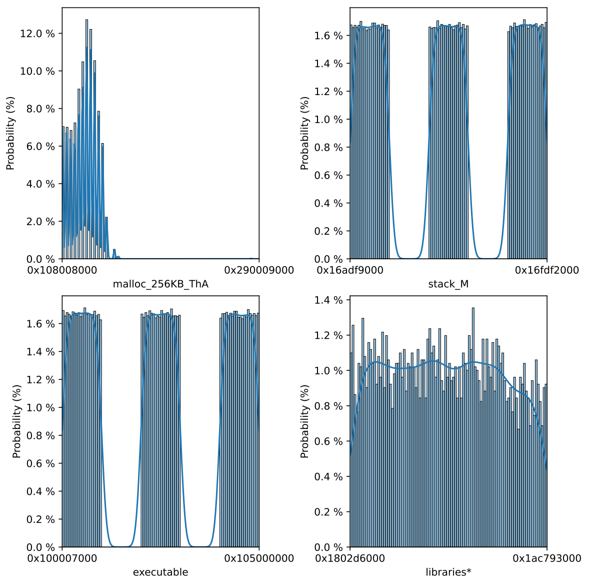

Figure 5: Probability Distribution in MacOS M1 Native. All the distributions are available at: https://zenodo.org/records/12968870/files/macos_native_distribution.pdf

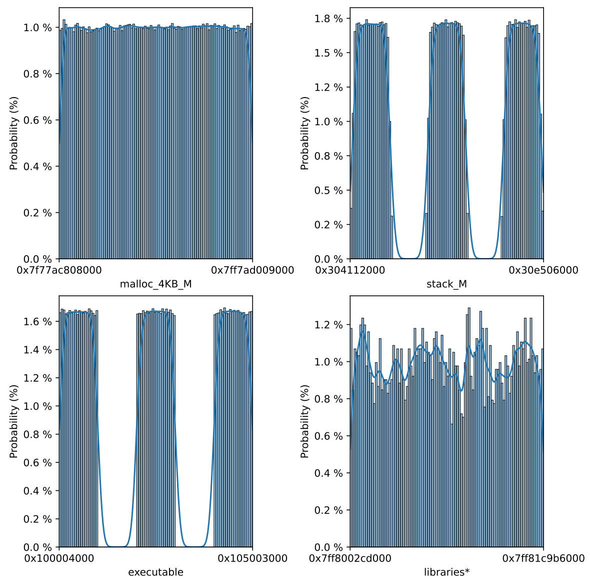

Figure 6: Probability Distribution in MacOS M1 Rosetta. All the distributions are available at: https://zenodo.org/records/12968870/files/macos_rosetta_distribution.pdf

contains a lot of useful gadgets and functions, it is a common target for attackers to build exploits.

Further inspection of complete data in Table 4 highlights the low entropy of threads. We know from the documentation of `pthread_create()` [?] that the default size of the thread stack is `2MB` ; being the thread stack allocated with `mmap()` function we see that the entropy of the `2MB` thread stack and the `4MB` `malloc()` are very similar, so this is the regular behavior.

Correlation Entropy. As expected from the allocation layout, many objects exhibit a high correlation with other objects. In particular, for both kernel versions, the Correlation Entropy between the executable and the heap is 13 bits, lowering their entropy by 15 bits; in other words, a leak can reduce the attack effort by 32,000 times. Moreover, all mapped areas suffer from consecutive allocation. Finally, libraries allocated with Linux Folios (*lib_big*) and libraries allocated with standard pages (*lib_small*) are no longer contiguous. Instead, they have 9 bits of Correlation Entropy. This is a positive side effect since the position of other libraries changes with every execution, but the Correlation Entropy is extremely low and can be easily bruteforced.

5.2 MacOS

We analyze Mac OS M1 Ventura 13.4.1 on the Native Apple M1 (ARM) processor and on x86_x64 using the Rosetta framework. MacOS is a system that randomizes libraries only at boot, so we reboot the machine several times to collect the samples. For each reboot, we collect 500 samples, and we restart the machine 5,500 times, thus obtaining 2,750,000 runtime samples and 5,000 boot-time samples. Additionally, the sampling program is compiled with *Clang* as *Position Independent* (flag `-fPIE`) and relies on external libraries, i.e., not compiled with the flag `-static`. Unlike Linux, macOS does not offer an option to adjust the level of randomization; therefore, we rely on the OS's default randomization settings.

Allocation Layout. The allocation on MacOS using Native ARM architecture, visible in Figure 3, are mainly grouped among low addresses with only very small allocations that belong to the so-called `MALLOC_NANO` area on the higher end of the memory. This suggests a high correlation between objects located at low addresses. In Figure 4 instead, we can see that the allocation layout on MacOS using Rosetta is divided into four regions. The lower one locates explicitly mapped pages, the executable, and the shared memories. The second one locates the stack of main and threads, while the third one corresponds to the higher of the Native system, accommodating `MALLOC_NANO` and `TLS` variables. The higher one is dedicated to `malloc()` objects and libraries that, this time, present more visual distribution. Although we should expect a higher entropy, given the allocation ranges, the allocation of these objects is fragmented, which means that the alignment of these objects is much higher than a single page(`16KB`). This can result in a lower entropy.

Probability Distribution. On MacOS using Native ARM architecture, there are some uniform allocations regarding the `TLS` objects and `MALLOC_NANO` objects. Other sections have distributions characterized by high spikes or large groups, indicating a very low entropy. Figure 5 shows the probability distribution of the memory objects. The histogram plot is divided into 100 bins; in the uniform distribution, the probability of each bin is 1%. We can see in Figure 5 spikes up to 14% for malloc of `4MB`. Other objects (executable and stack) have a spotted uniform distribution. We can clearly see some gaps in the distribution; these are the probability of some addresses being higher than others.

In some sections, the randomization is probably linked to the intrinsic, not deterministic positioning of allocation and not to

Table 5: MacOS (Ventura 13.4.1) Absolute Entropy.

Object	Entropy		Object	Entropy		Object	Entropy		Object	Entropy	
	Native	Rosetta		Native	Rosetta		Native	Rosetta		Native	Rosetta
env	12.903	15.972	malloc_16B_M__1	12.581	12.609	malloc_16B_ThA__1	13.725	13.372	malloc_16B_ThB__1	14.050	13.514
stack&argv_M	12.028	15.121	malloc_512B_M__1	7.551	16.548	malloc_512B_ThA__1	9.237	17.710	malloc_512B_ThB__1	9.394	18.169
shared_memory	11.583	13.584	malloc_4KB_M__1	7.465	16.502	malloc_4KB_ThA__1	8.484	17.412	malloc_4KB_ThB__1	8.710	17.628
tls_M	12.549	14.668	malloc_256KB_M__1	3.189	12.000	malloc_256KB_ThA__1	4.550	13.196	malloc_256KB_ThB__1	4.755	13.398
libraries*	15.625	14.894	malloc_4MB_M__1	3.231	12.059	malloc_4MB_ThA__1	5.038	13.600	malloc_4MB_ThB__1	5.159	13.942
executable	11.583	13.583	malloc_128MB_M__1	7.106	16.070	malloc_128MB_ThA__1	7.240	16.093	malloc_128MB_ThB__1	7.485	16.095
stack_ThA	11.583	14.672	mmap_single_M__1	11.587	13.583	mmap_single_ThA__1	12.478	13.585	mmap_single_ThB__1	12.454	13.585
tls_ThA	13.159	14.333	malloc_16B_M__2	12.636	12.692	malloc_16B_ThA__2	14.516	14.188	malloc_16B_ThB__2	14.551	14.573
stack_ThB	11.583	14.672	malloc_512B_M__2	7.619	16.615	malloc_512B_ThA__2	9.984	18.438	malloc_512B_ThB__2	9.939	19.102
tls_ThB	13.519	14.721	malloc_4KB_M__2	7.533	16.579	malloc_4KB_ThA__2	9.137	18.032	malloc_4KB_ThB__2	9.180	18.489
			malloc_256KB_M__2	3.274	12.128	malloc_256KB_ThA__2	5.282	13.844	malloc_256KB_ThB__2	5.349	14.256
			malloc_4MB_M__2	3.278	12.132	malloc_4MB_ThA__2	5.295	13.858	malloc_4MB_ThB__2	5.360	14.277
			malloc_128MB_M__2	7.857	16.041	malloc_128MB_ThA__2	6.650	16.327	malloc_128MB_ThB__2	7.162	16.402
			mmap_single_M__2	12.046	13.583	mmap_single_ThA__2	12.611	13.585	mmap_single_ThB__2	12.582	13.585

explicit ASLR action. Rosetta does a good job distributing the allocations, with all `malloc()` objects having a uniform distribution. The remaining objects have a uniform distribution that is not contiguous. The reader can see the mentioned in Figure 6. Finally, on both systems, libraries have a uniform distribution even if it is not visible due to the smaller amount of reboot samples.

Absolute Entropy. As we can see from Table 5, the overall entropy is low, with poor randomization on executable objects. Libraries have an insufficient boot-time randomization entropy (12.2 bit), while the executable has a worse runtime randomization entropy (11.5 bit) in Native ARM and slightly better entropy (13.5) using Rosetta. Moreover, with the Native system, the objects allocated with `malloc()` are very poorly randomized, ranging from a maximum entropy of 12 bits to a minimum entropy of 3 bits for the main thread. Other objects and `mmap()` allocated objects achieve a lower entropy around 12 bit as well, which is still relatively low. Finally, thread allocation entropies are slightly better than the main thread ones. Instead, Rosetta entropies are overall higher but still insufficient and again particularly low in objects allocated with `malloc()` and `mmap()`. One reason for such a low entropy is the size of Apple M1 single pages. In fact, `16KB` pages require an alignment of 14 bits, i.e., only 33 bits instead of 35 can be used to randomize an address. Moreover, the allocations slots for these allocations are surprisingly at 1 to `4MB` (20 to 22 bits) of distance from one another, which confirms our hypothesis in Figure 4.

Correlation Entropy. In the Native MacOS, we have low Correlation Entropies due to the several objects allocated very close to each other. Although they are low, there is no need to have any prior knowledge since absolute entropies are as low as correlated ones, thus making bruteforce directly on the target object the preferred strategy. Instead, for Rosetta, we have four different memory regions, and we have higher Correlation Entropies. In particular, we have high Correlation Entropies between the objects of different regions. However, as in the Native MacOS system, the absolute entropies are so low that a direct bruteforce on the desired object is preferred.

5.3 Windows 11

Windows 11, like MacOS, has objects that are randomized at boot-time. These objects are the libraries and the executable. Hence, we collect 4,000,000 runtime samples and 20,000 boot-time samples. We compile our sample script with the *MSVC* compiler, employing the flags `/DINAMICBASE` and `/HIGHENTROPYVA`. At the kernel level, we rely on the default ASLR configurations, which are `High-entropy ASLR` and `Bottom-up ASLR`. These settings are intended to provide the highest level of ASLR effectiveness in Windows.

Allocation Layout. Looking at the allocation layout in Figure 7 we clearly identify two regions. The first one is located at low addresses and contains all the `malloc()` and `VirtualAlloc()` objects, as well as threads' stacks. At the other end of the memory, we can see the executable and the libraries, which are the objects whose position is determined at boot-time.

Probability Distribution. Figure 8 shows probability distributions for Windows. Here, we can observe three different shapes for the distribution. We have threads and stacks that are randomized almost uniformly; the lowest addresses are more likely to be used. The executable and libraries are also randomized uniformly, but given the lower amount of sample (they are randomized at boot-time), this is not clear from Figure 8. Finally, we have `malloc()` and `VirtualAlloc()` objects that are randomized following a triangular distribution. This last aspect usually means that the position is obtained by combining two independent sources of entropy, obtaining an Irwin-Hall distribution. This choice, even if it provides a larger entropy than the single random variable, potentially exposes the system to attacks regarding the most common value, reducing the absolute effort needed to de-randomize the section.

Absolute Entropy. From Table 6, we see an Absolute Entropy greater than 23 bits overall, with some objects reaching 31 bits. As expected, Large pages have less entropy compared to other sections due to a higher page alignment. However, we have low absolute entropies for boot-time randomized objects, i.e., the executable and the libraries. This poses a major risk to the security of the system since these objects are the most used for control-flow hijacking. Hence, bruteforce attacks are feasible in a short period of time.

Correlation Entropy. Simirarly to the other OSes, we have what expected from the allocation layout: A high correlation inside the identified regions and a low correlation between regions. However, no relevant correlations emerge as the absolute entropies of relevant objects are lower than correlated ones.

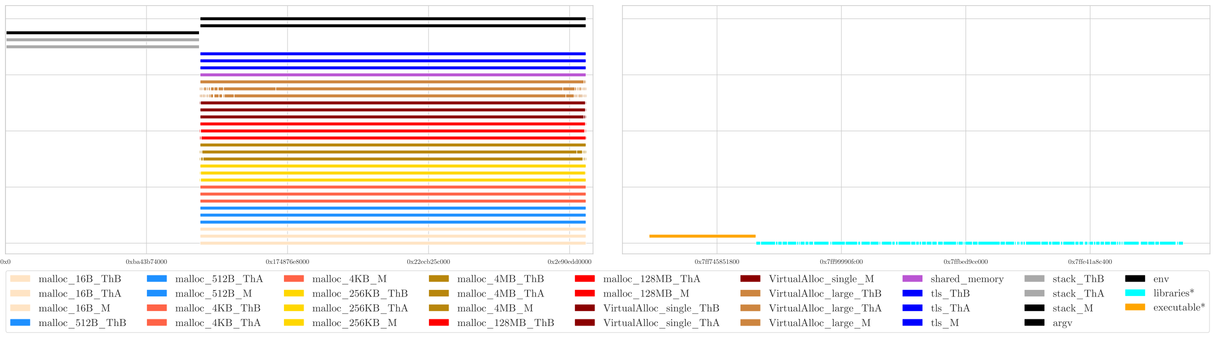

Figure 7: Windows 11 Allocation Layout.

Table 6: Windows 11 Absolute Entropy.

Object	Entropy	Object	Entropy	Object	Entropy	Object	Entropy
env	25.914	malloc_16B_M__1	30.818	malloc_16B_ThA__1	30.921	malloc_16B_ThB__1	30.822
argv	25.416	malloc_512B_M__1	25.914	malloc_512B_ThA__1	31.558	malloc_512B_ThB__1	30.837
stack_M	28.151	malloc_4KB_M__1	25.914	malloc_4KB_ThA__1	28.961	malloc_4KB_ThB__1	30.368
shared_memory	25.345	malloc_256KB_M__1	25.359	malloc_256KB_ThA__1	25.881	malloc_256KB_ThB__1	26.879
tls_M	25.359	malloc_4MB_M__1	29.250	malloc_4MB_ThA__1	24.755	malloc_4MB_ThB__1	24.656
libraries*	18.966	malloc_128MB_M__1	25.528	malloc_128MB_ThA__1	23.057	malloc_128MB_ThB__1	23.153
executable*	16.985	VirtualAlloc_single_M__1	25.243	VirtualAlloc_single_ThA__1	25.158	VirtualAlloc_single_ThB__1	25.185
stack_ThA	27.407	VirtualAlloc_large_M__1	19.654	VirtualAlloc_large_ThA__1	19.002	VirtualAlloc_large_ThB__1	19.102
tls_ThA	30.808	malloc_16B_M__2	30.796	malloc_16B_ThA__2	31.852	malloc_16B_ThB__2	31.845
stack_ThB	27.407	malloc_512B_M__2	28.872	malloc_512B_ThA__2	31.311	malloc_512B_ThB__2	31.487
tls_ThB	30.780	malloc_4KB_M__2	28.931	malloc_4KB_ThA__2	28.937	malloc_4KB_ThB__2	30.751
		malloc_256KB_M__2	28.931	malloc_256KB_ThA__2	27.407	malloc_256KB_ThB__2	26.749
		malloc_4MB_M__2	24.122	malloc_4MB_ThA__2	23.838	malloc_4MB_ThB__2	24.014
		malloc_128MB_M__2	23.340	malloc_128MB_ThA__2	22.394	malloc_128MB_ThB__2	22.538
		VirtualAlloc_single_M__2	25.209	VirtualAlloc_single_ThA__2	25.079	VirtualAlloc_single_ThB__2	25.098
		VirtualAlloc_large_M__2	19.466	VirtualAlloc_large_ThA__2	18.634	VirtualAlloc_large_ThB__2	18.749

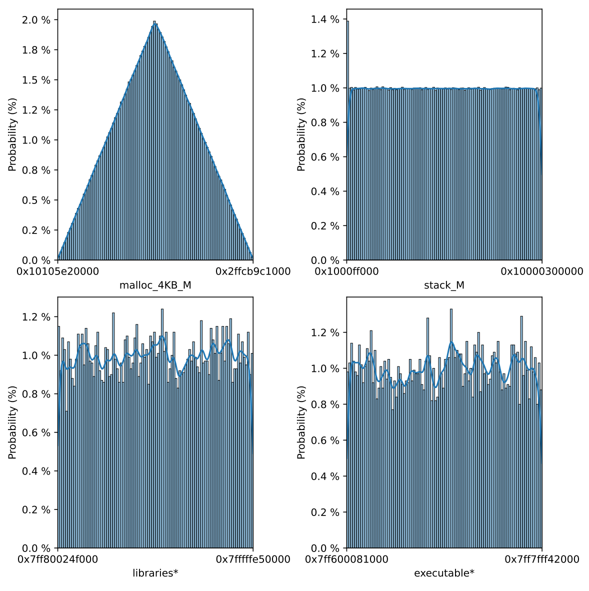

Figure 8: Probability Distribution in Windows 11. All the distributions are available at: https://zenodo.org/records/12968870/files/windows_distribution.pdf

6 PRACTICAL ATTACKS TO ASLR

While it is straightforward to defeat ASLR when there are memory leaks, the practicality of attacks without direct memory leaks must be discussed. We consider the following attacks: ① *bruteforce*, ② *spraying attacks*, ③ *crossections*, and ④ *partial overwrite* attacks.

For these attacks, we mainly consider CVEs and security articles related to Linux for several reasons. Among all the OSes considered, Linux is the only completely open-source, which facilitates the development of open-source software. This aids the discovery and analysis of vulnerabilities, as security analysts and researchers can perform analyses starting from the source code and can rely on state-of-the-art security tools developed by the community. Additionally, most commonly used applications and libraries on Linux are well-documented, which favors the development of PoCs and the creation of security articles, making them easier to follow and understand. Finally, Linux is the OS with the most disclosed vulnerabilities [?]. Hence, we can find more examples of attacks and vulnerabilities to analyze.

As memory corruptions happen in all the OSes, the attacks we discuss in this section are not exclusive to Linux, but they can be applied to other OSes as well. However, the feasibility of the exploitation must also consider all the other mitigations in place in the OS. Nonetheless, if they are not sufficient to prevent exploitation, our findings in ASLR implementations highlight weaknesses that must be addressed.

In the following attacks, we consider an attacker capable of performing 300 tps as discussed in Section 2.6. We remind the reader that this value is realistic for both local and remote scenarios since an attacker can parallelize over several threads/processes and connections, respectively.

Bruteforce Attacks. This is the most naive attack to ASLR. The attacker guesses the position of a memory object and executes the exploit with hardcoded positions. This is the primary threat that ASLR is designed to protect. Depending on the entropy of the memory object, the number of attempts to guess the position of the object varies. We argue that the entropy of the memory object in modern systems is not high enough, making this type of attack feasible in practice. To prove our point, we demonstrate how the lower entropy generated by Linux Folios can be exploited in recent vulnerabilities (CVE-2021-3156) [?][1] to perform an attack in a reasonable amount of time (i.e., minutes). In particular, this CVE reports a heap overflow vulnerability on `sudo`, from version `1.8.2` (included) to `1.9.5p2` (excluded). Therefore, we set up an Ubuntu 22.04 environment, with kernel version 6.4.9 and `sudo 1.9.4`. In this setup, the glibc is allocated through Linux Folios. Hence, its memory page has a randomization entropy of only 19 bits.

We develop a Proof of Concept (PoC)[2] such that the overflow modifies a function pointer being called later in the execution. The new function pointer is a hardcoded glibc address that contains a gadget which performs a stack pivoting attack, moving the stack pointer close to the environment variables, where we place a ROP chain to achieve privilege escalation. The success rate of the exploit is directly influenced by the effectiveness of ASLR in randomizing glibc position. A robust ASLR implementation would render our exploit ineffective. However, the reality is that ASLR randomization does not entirely prevent the exploit's success.

To measure the average time taken and to verify the entropy value, we run the exploit 500 times. Moreover, given that an attacker can also parallelize the exploits, we run 8, 4, and 2 exploits in parallel. Theoretically, with an entropy of 19 bits, the average number of attempts to guess the position of the glibc is $2^{19} = 524,288$ attempts. As shown in Table 7, we achieve an average tps of 283 and an average number of attempts of $555,765$, which is very close to the expected value. Moreover, we can see that even with one exploit, the time to achieve privilege escalation is 32 minutes and 44 seconds, which is very low. It is even lower when 8 exploits are run in parallel. In this case, the first exploit achieving privilege escalation takes only 4 minutes with an average of $58,776$ attempts. In addition, all the exploits running in parallel generate an average tps of 1,904, underlining the possibility of achieving a higher number of tps w.r.t. the reference value of 300 tps considered in Section 2.6.

This PoC highlights the problems arising from low Absolute Entropy: relevant memory objects like the libraries and the executable should have a high Absolute Entropy since these regions are very often needed by attackers to obtain command execution. Therefore, the low Absolute Entropy of the glibc in Linux Folios is a significant security risk. We believe that the entropy of relevant objects should be increased to a minimum of 24 bits (i.e., $2^{24} = 16,777,216$) which can be bruteforeced in more than a day with 300 tps on average. Instead, we consider a good entropy value to be 30+ bits, which would require more than a month to be bruteforced.

[1]The same vulnerability also affects MacOS Big Sur 11.2, MacOS Catalina 10.15.7, and MacOS Mojave 10.14.6 [?]

[2]https://zenodo.org/doi/10.5281/zenodo.12784286

Table 7: Average attempts and time for privilege escalation.

# Parallel Exploits	Avg Attempts	Avg Time	Avg TPS
8	58,776	00:04:06::904	1,904
4	128,884	00:08:23::539	1,023
2	275,585	00:16:51::781	544
1	**555,765**	**00:32:44::597**	**283**

Spraying Attacks. Even when the absolute entropy is high (>27 bits), *Spraying Attacks* can drastically reduce it by repeating the same data for several memory pages. This attack aims to repeat the same payload across several memory pages to increase the likelihood of successfully referencing the pointer.

Recently, a vulnerability on the glibc (CVE-2023-4911, a.k.a. Looney Tunables) has been exploited using a stack spraying technique to achieve local privilege escalation [?]. The attack exploits a buffer overflow vulnerability on the Linux dynamic loader `ld.so` to change the string pointer of `l_info[DT_RPATH]`, a string that specifies the libraries search path. In particular, the attack modifies the string pointer to point to the environment variables, where a string representing an attacker-controlled directory resides. Under Linux, the environment variables have 24.5 bits of entropy. This value does not guarantee strong randomization since it requires an average of $2^{24.5} = 23,726,566$ attempts to locate the precise address of the environment variables. However, repeating the same string across several memory pages increases the likelihood of successfully referencing the pointer. Given that the maximum size for environment variables in Linux is `6MB`, and if the same string spans multiple pages, the likelihood of successful exploitation improves to 1 over 4,096 (i.e., $\frac{24GB}{6MB}$). This means that, at a rate of 300 tps, local privilege escalation is achieved in about 14 seconds.

Spraying attacks are feasible in any memory region an attacker can control exhaustively. For instance, other objects that can be used for spraying attacks are all the dynamically allocated pages, such as the heap. In Linux, the heap has an entropy of 28.8 bits, which is the maximum entropy in the system. Nonetheless, this may not deter an attacker capable of allocating gigabytes of memory pages without a limit on heap growth. To make these attacks less relibale, user-controlled memory pages should have a higher absolute entropy (>30 bits).

Crossections Attacks. When the entropy is high enough, bruteforcing a precise memory location may become unrealistic. However, if there are *memory leak* vulnerabilities, the attacker may rely on this information to exploit the correlation between memory objects [? ?]. As we pointed out in Section 5, strong correlation paths exist between objects that are closely allocated in memory. For instance, the executable and heap entropy in Linux are 28.8 bits. Hence, the average number of attempts to bruteforce a precise address of one of the two objects is $2^{28.8} = 467,373,275$ attempts, or $1,557,911$ seconds (18 days) if we consider 300 tps. However, with a memory leak on the heap, an attacker can bruteforce a precise address of the executable in $2^{13} = 8,192$ attempts, on average, which is approximately 27 seconds.

Although this example highlights the risks of having a strong Correlation Entropy between two memory objects, this is less relevant than having a low absolute entropy; a weak Correlation Entropy between two memory objects is useless if the two objects have weak Absolute Entropies as well. This happens, for instance, on MacOS.

Partial Overwrite Attacks. In some situations, an attacker can partially overwrite a function pointer to hijack the control flow, potentially compromising the security of a system. For instance, by changing the lower bits of a function pointer it is possible to call a different function within the same library, or a function that belongs to relatively close, or contiguous, libraries. For this reason, a good ASLR implementation should allocate libraries with a very high Correlation Entropy one another. This is not the case for Linux, where, during program startup, all the libraries are allocated contiguously or with low entropy to one another. Therefore, whenever a vulnerability allows a partial overwrite of a function pointer, the difference between the original and the targeted one is always the same. However, ASLR changes most of the bits of the addresses and thus, the chances of correctly modifying the lower bits of the address depend on the distance between the two pointers and the page alignment. For instance, with a page alignment of `4KB` and a pointers difference of less than 256 bytes, the attack is completely deterministic. Instead, with a pointers difference lower than a standard page (`4KB`), the attack requires an average of 16 attempts (4 bits of randomness). Finally, with a pointers difference higher than a standard page, the attack requires an average of 4,096 attempts (12 bits of randomness). However, with Linux folios, libraries with a size larger than `2MB`, such as the `glibc`, have a page alignment of `2MB`. Therefore, whenever the pointer difference is lower than `64KB`, the attack may be completely deterministic; if the targeted pointer belongs to the same library as the original one, the attack is completely deterministic. Otherwise, there might be at least 9 bits of entropy due to the strong correlation between libraries allocated with Folio and standard pages.

This attack should highlight two major problems in ASLR implementations: ① libraries should never be contiguous with one another, but instead, they should be independent (high Correlation Entropy), and ② higher page alignments drastically reduce the chance of exploitation.

7 LONG LIFE TO ASLR

As we mentioned in Section 5, modern ASLR implementations have several flaws. MacOS is characterized by a low absolute entropy overall, with entropy for relevant sections such as the executable and the libraries ranging from 11.5 to 15.5 bits. Windows 11 has better peaks, reaching up to 31 bits of entropy, but it also has a poor randomization where it matters the most: the executable and the libraries, reaching at most 19 bits of entropy. Instead, Linux achieves high entropies in all the relevant objects, including the stack and the heap, resulting in the best OS in randomization. However, the newest kernel versions and memory leaks can considerably reduce the effort of bruteforce attacks. Ideally, any memory object in the system should reach an absolute entropy of 31 bits, such as one of the small allocated objects in Windows 11, to consider the system resilient against bruteforce attacks. If we consider 300 tps and an average of 2 billion attempts (2^{31}), it would require an average of approximately 83 days to guess the position of an object in memory, which is not practical, especially if Intrusion Detection Systems (IDSs) are in place.

Therefore, our first proposal consists of a virtual memory fragmentation where each relevant object has its own region. For instance, in Linux, the addresses below the executable and the heap are not used. Hence, there is a place for objects, like the executable or the heap, also to resolve the correlation between these two objects. Similarly, libraries can be moved to lower addresses to remove the correlation between this latter and allocated pages.

Secondly, we propose the utilization of 5-level page tables so that the available bits in an address extend from 48 to 57. We already have processors that support 5-level page tables, but modern systems seem not to adopt them. The reason is probably related to performance once more; one memory access to virtual memory results in six accesses to physical memory instead of five, thus slightly increasing the memory access time.

8 CONCLUSIONS

In this paper, we conduct the first comprehensive evaluation of ASLR effectiveness across major platforms through the statistical analysis of memory object positions. Adopting a low-bias estimator, the NSB estimator, allows us to reduce the necessary sample size, thereby facilitating the analysis of reboot randomization, a notably time-intensive operation.

We highlight significant weaknesses of current implementations of ASLR, like the lack of entropy of libraries and executable objects in Windows and MacOS or the entropy reduction found in recent Linux distributions (introduced with Folios). Overall, Linux distributions provide the best randomization- still insufficient to stand against modern exploitation techniques - while Windows and MacOS fail to randomize key memory areas like executable code and libraries adequately. Our findings highlight opportunities for OS vendors to strengthen implementations and better protect users from malicious attacks. Addressing reduced entropy from correlations, optimizing allocation patterns, and increasing object granularity could all fortify defenses. Moreover, this research suggests that the evolution of operating systems is often not security-focused, and the introduction of Linux Folios confirms this claim. We expect major changes with the broad adoption of a 5-level paging system, providing by construction more bits to the randomization process [**?**]. Another aspect highlighted in this work is the role of allocation patterns and the difficulty of correctly modeling such behavior. Real-world software is a complex ecosystem of interacting objects, and their performance may vary significantly from expectations and often be lower.

9 ACKNOWLEDGEMENTS

Lorenzo Binosi acknowledges support from TIM S.p.A. through the PhD scholarship. Additionally, this work was partially supported by Project FARE (PNRR M4.C2.1.1 PRIN 2022, Cod. 202225BZJC, CUP D53D23008380006, Avviso D.D 104 02.02.2022) and Project SETA (PNRR M4.C2.1.1 PRIN 2022 PNRR, Cod. P202233M9Z, CUP F53D23009120001, Avviso D.D 1409 14.09.2022). Both projects are under the Italian NRRP MUR program funded by the European

Union - NextGenerationEU. Finally, this work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.