POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

The Illusion of Randomness: Demystifying the Entropy of ASLR on Common Operating Systems

Tesi di Laurea Magistrale in
Ingegneria Informatica

Author: **Gregorio Barzasi**

Student ID: 977551
Advisor: Prof. Mario Polino
Co-advisors: Lorenzo Binosi
Academic Year: 2022-23

Abstract

Address Space Layout Randomization (ASLR) is a crucial defense mechanism employed by modern operating systems to mitigate exploits by randomizing the memory layout of processes. However, real-world implementations of ASLR are imperfect and subject to weaknesses that can be exploited by attackers. This thesis evaluates the effectiveness of ASLR on major desktop and mobile platforms, including Linux, MacOS, Windows, and Android. An analysis tool was developed to take samples of memory object addresses across multiple processes, threads, and system reboots; statistical analysis was performed on the sampled data to quantify the entropy, or randomness, of object placement; memory layouts were also analyzed to identify correlations between objects that could reduce overall entropy. The results show that while some systems like Linux distributions provide strong randomization, desktop platforms like Windows, MacOS and mobile platforms like Android often fail to adequately randomize key areas like executable code and libraries. Moreover, a major entropy reduction in the entropy of libraries after the Linux 5.18 version was discovered. Positive Correlation paths that an attacker could leverage to significantly reduce exploitation complexity were also identified. In the end, found weaknesses were ranked based on severity, and a proof-of-concept attack validated our entropy estimates. The findings provide insights into each platform's resistance to memory exploitation techniques. They also highlight opportunities for OS vendors to strengthen ASLR implementations against these common vulnerabilities. Future work includes analyzing hardened and custom operating systems, quantifying the impacts of optimization techniques, and developing a profiling tool to analyze real-world executables.

Keywords: ASRL, entropy evaluation, operating system security, address randomization effectiveness

Abstract in lingua italiana

L'Address Space Layout Randomization (ASLR) è un importante meccanismo di difesa impiegato dai moderni sistemi operativi per mitigare la possibilità di exploit randomizzando la disposizione della memoria dei processi. Tuttavia, le implementazioni reali di ASLR sono imperfette e soggette a debolezze che possono essere sfruttate dagli aggressori. Questa tesi valuta l'efficacia dell'ASLR sulle principali piattaforme desktop e mobili: Linux, MacOS, Windows e Android. È stato sviluppato uno strumento di analisi per campionare indirizzi di oggetti in memoria attraverso più processi, thread e riavvii del sistema. I dati campionati sono stati analizzati statisticamente per quantificare l'entropia, o randomicità, del posizionamento degli oggetti. Sono stati analizzati anche i layout di memoria per identificare le correlazioni tra gli oggetti che potrebbero contribuire a ridurre l'entropia complessiva. I risultati mostrano che, mentre alcuni sistemi come le distribuzioni Linux generalmente forniscono una forte randomizzazione, le piattaforme desktop come Windows e MacOS, e le piattaforme mobili come Android spesso non riescono a randomizzare adeguatamente aree chiave come il codice eseguibile e le librerie. Inoltre, è stata scoperta un'importante riduzione dell'entropia delle librerie dopo la versione 5.18 di Linux. Sono stati identificati anche percorsi di correlazione positiva che un aggressore potrebbe sfruttare per ridurre significativamente la complessità di attacco. Alla fine, le debolezze trovate sono state classificate in base alla gravità e un attacco proof-of-concept ha convalidato le nostre stime riguardo l'entropia. I risultati forniscono indicazioni sulla resistenza di ciascuna piattaforma alle tecniche di attacco. Inoltre, evidenziano le opportunità per i produttori di sistemi operativi di rafforzare le implementazioni ASLR contro queste vulnerabilità comuni. Il lavoro futuro prevede l'analisi di sistemi operativi rinforzati dal punto di vista della sicurezza, la quantificazione dell'impatto delle tecniche di ottimizzazione delle prestazioni e lo sviluppo di uno strumento di profiling per analizzare gli eseguibili del mondo reale.

Parole chiave: ASRL, valutazione entropia, sicurezza di sistemi operativi, randomizzazione degli indirizzi

Contents

1 | Introduction

In the context of software exploitation the more information an attacker can gather about the targeted system, the easier it is to reach the goal of exploiting it. In particular, one crucial piece of information for the success of *buffer overflows attacks* (as many others) is the exact address of objects inside the memory space of the running program; this information could allow an attacker to successfully exploit the software vulnerabilities to modify memory content, leak information, or even deviate the execution flow of the program, leading in the worst case scenario to arbitrary code execution. Even if this type of vulnerability are very well known, they still account for almost 20% of the CVE reported [18].

Address Space Layout Randomization (ASLR) is a security measure developed to improve resilience against exploitation techniques that need precise memory location in order to work. 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, we would like ASLR to randomize each and 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 we will see in the next sections on many systems available at the moment 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 in some way: we call this *Positive Correlation* [13].

Even if *Positive Correlation* it is an already known vulnerability [6], the research available at the moment lacks a broad estimate about the severity of this vulnerability in terms of entropy reduction; they are also limited in terms of section analyzed and the operating systems considered [14] as they are strongly pointed towards Linux enterprise solutions; moreover, available research are not up to date with the recent introduction of ARM

architecture in Apple devices, so the performance of ASLR on those systems is completely unknown.

In this work we tried to fill these gaps, in particular those are our contribution:

- We developed a tool capable of analyzing ASLR implementation of commercial operating systems (Linux, Windows, MacOS and Android) by monitoring many memory objects allocated in a multi-threaded fashion, and reducing the number of samples needed to perform the analysis by adopting *NSB* entropy estimator.
- We analyzed the Absolute Entropy and Correlation Entropy of Linux 5.17.15, Linux 6.4.9, Windows 11, MacOS M1, MacOS M1 with Rosetta, and Android 13.
- We confirmed that on Linux, a leak of a Heap address still reduces the entropy of the text section through Positive Correlation from 27.4 bit to just 13 bit thus reducing the attack complexity from 178 million attempts to just 8 thousand [13]; we now found the same Positive Correlation path for Heap and Libraries with 9 bit of entropy.
- We discovered a sudden reduction in randomization entropy of shared libraries in Linux systems since the 5.18 release due to the introduction of Linux Folios performance optimization.

The thesis is organized as follows:

- Chapter 2 contains a review of ASLR implementation and their limitations. An overview State-of-the-art analysis tool and how we plan to improve the analysis tecnique.
- Chapter 3 discuss the approach and architecture of the developed tool based on the NSB entropy estimator.
- Chapter 4 presents the actual implementation along with the challenges faced during the development.
- Chapter 5 discuss the results of analysis on the different OS, comparing their performance before and after the rebooting process.
- Chapter 6 propose some attack scenario based on the results presented in previous chapters along with a POC attack to evaluate empirically the results.
- Chapter 7 discuss conclusions and future research directions.

2 | Background and Motivation

Analyzing ASLR performance is crucial as it is one of the last security measures that need to be defeated before exploitation of a system. It's fair to say that it isn't the only security measure in defense of attacks, in fact, we can rely also on *NX* bit and *Stack Smashing Protection (SSP)* to be present in case a new bug or vulnerability is discovered and our system is in danger to be compromised; however, the fact that we can rely on other security systems, isn't an excuse for vendors to adopt broken ASLR implementation; moreover, being aware of the peculiar OS limitations can provide useful insight during the choice of which operating system we shall use, according to the requirement and threat model of our situation.

In the following sections, we will discuss the most common weaknesses, justify our operating system choice, and present an overview of the state-of-the-art in terms of ASLR evaluation.

2.1. Performance Metrics

To understand the limitations of ASLR implementation we shall start underlining the goals and the expected performance of such a system: increase the number of attempts an attacker should perform on average to correctly guess a memory address, through the obfuscation of memory layout; to do so we would like to have a system that randomizes memory objects often, with low predictability and with high granularity. To be more rigorous we can use the Taxonomy of ASLR implementations proposed by Marco-Gisbert and Ripoll [14] that categorizes the capabilities of ASLR implementation into three categories: when, what, and how. Based on the performance in these three categories we can evaluate how well an ASLR implementation behaves and what should be improved.

2.1.1. When

ASLR systems in different OS differentiate themself by the frequency on which they renovate the randomization of their memory section. We aim to have a system that

randomizes every object in memory, at the moment they are allocated; this presents many difficulties, both in terms of performance reduction and increased complexity of implementation so, at the moment, none of the commercial OS implements such advanced technique. The best we can aim for, is to have the memory randomized every time a new process is allocated in memory (as in the case of Linux implementation) however many OS randomize only some part of memory and only at boot; this is considered to be a broken implementation of ASLR, as a local attacker could use information gathered from other processes to precisely localize library position; moreover, a remote attacker could just brute force the address averaging half of the original entropy, or exploit byte-for-byte attacks to reduce attack complexity even more [29].

2.1.2. What

One more aspect regards both the granularity of what is randomized and the number of sections randomized: even one single non-randomized object can be exploited to take over a system. For example, some implementation does not randomize all executable objects or it does that only at boot, strongly reducing the overall security of the system and increasing the chance of success of a ret2lib or ROP attack.

2.1.3. How

The last category regards how objects are randomized, in terms of the number of bits randomized and the relative position of the objects. In fact, the current implementation of ASLR utilizes only so-called **partial-VM** randomization, where the virtual memory is divided into sections and then objects are placed inside them; this strategy increases the chances of having Positive Correlation paths.

2.2. Implementation Weakness

All aspects mentioned in Section 2.1 affect unpredictability, and we can measure that mathematically using entropy. In particular, the lack of Absolute Entropy and Correlation Entropy is the most common weakness related to ASRL implementations.

2.2.1. Low Absolute Entropy

The major problem affecting ASLR implementations is low Absolute Entropy, that in this context is directly linked to the brute force effort that an attacker needs to put in to correctly guess directly the position of a memory object, without using any particular

technique to predict the position. In theory, the limit of Absolute Entropy of an object is the size of the address, so 64-bit for the system considered in this research; however, the practical limit is way lower, as the `N` *Most Significant bits* (`MSb`) are often not implemented at a hardware level, and in software appear as a fixed value, or 0. For example, in 4-level paging implementations of Linux kernel the `17 MSb` are sign extensions of the address and even in the most recent implementation with 5-level paging the practical limit for ASLR is 56-bit [25].

To address content inside memory pages, page offsets are used and are represented by `N` fixed *Least Significant bits* (`LSb`), thus the size of pages impacts directly the number of bits available to the randomization process. For the most common used page size of 4 KB, we have the `12 LSb` address fixed; this problem is even more relevant when we use huge pages (eg. `2MB` on Linux, so `21 bit` of page offset).

Because of the fixed parts of addresses, the updated maximum entropy achievable for a 64-bit system using 4-level paging and pages of 4KB is $47bit - 12bit = 35bit$. Unfortunately, no 64-bit OS comes even close to this performance as other factors are involved in the estimation of entropy.
Some of them are:

- **Presence of growable sections:** reduces the possibility of placing this kind of object in memory, as they need space to grow up or down (such as heap and stack).
- **Memory fragmentation:** the allocation and de-allocation of objects may reduce the probability of finding free continuous blocks.
- **Not uniformity:** the probability distribution inside the sections of the Virtual Memory is a relevant aspect. If it's not uniform an attacker could target the most common value to increase the chances of success after doing some dynamical analysis on the executable.

2.2.2. Low Correlation Entropy

Many memory objects are correlated in various ways, so the information provided by the disclosure of a memory address sometimes can reduce significantly the complexity of an attack. We can quantify the reduction by looking at Correlation Entropy, which represents the entropy of the offset between two memory objects.

We can identify two cases:

- **Positive Correlation**: The offset between two memory objects has an entropy lower than the sections of origin, so a leak is a piece of useful information to de-randomize other sections.
- **Negative Correlation**: The offset between two memory objects has an entropy higher or equal than the sections of origin, thus, is easier to guess the original position than the offset, even in the presence of a leak. This is the case when dealing with two uncorrelated variables.

Not all Positive Correlation scenarios are problematic, in fact, if the Correlation Entropy is high enough we can consider the object to be secure, even if it's slightly correlated with other objects. The problem arises when the Positive Correlation is so severe that an address leak could be potentially used to brute force the position of other objects in a reasonable time, which in our case was considered to have 20 bit of entropy as we will explain in Section 5; when this happens, we can identify what we call a *Positive Correlation path* that can be potentially exploited.

The most severe case of Positive Correlation presents 0 Correlation Entropy because the offset is fixed. This issue was exploited in the famous `off2libc` attack [12] and led to the introduction of the Effective Entropy concept into ASLR-related discussion [6].

2.3. OS Choice Motivation

When we look at research related to ASLR analysis we can see a stable trend pointing 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 [27]. Enterprise servers require higher security standards than Consumer systems, so it's understandable that 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 counts for an insignificant 3% of the market share when compared with the 70% of Windows and 20% of MacOS systems [20]. Moreover, the recent adoption of ARM architecture by Apple made obsolete all the research regarding MacOS. When we look at mobile systems Android is by far the most common one with 70% [21] receiving less to no attention in recent research.

Because of this, we decided to focus our research on the consumer market considering the following:

- **Linux**: we choose Ubuntu because is by far the most common Linux distribution on the market [26].

- **MacOS**: we analyzed the newly commercialized ARM implementation, both using native compiled software and using the Rosetta framework.
- **Windows**: considered the last available version, 11.
- **Android**: considered the last available version, 13.

To perform our research we started from scratch with a new ASLR analyzer tool to better tackle the aspect we are interested in and to uniform the results over the considered OS.

2.4. ASLR Analysis Overview and Improvement

2.4.1. Linux and MacOS

The most advanced tool used in research is **ASLR-A** by *Marco-Gisbert and Ripoll* [13, 14]. 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 OS, 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 in particular when dealing with non-uniform distributions.
- Low sample dimension (only 1000 samples).
- Incorrect target identification (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 brute-forcing 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 [14] 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

OS considered. In fact, as mentioned in the previous section, ASLR performance is related to many runtime conditions 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's 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 MacOS platform.

2.4.2. Windows

For what concerns Windows, as far as we know, there are only three published research available. The first, regarding Windows Vista [28], is outdated, so we will focus only on the ones analyzing Windows 10 [5] and Windows 7 [2].

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 [5]. 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 [4].

The analysis of Windows 7 [2], even if is outdated and considered only 4 memory sections, concluded that the problems highlighted in Windows Vista [28] were still present, enforcing one more time the importance of doing this type of research.

2.4.3. Android

At the moment no quantitative research on Android randomization performance is available. Over the years many pointed out that PRNG on Android have low entropy [3, 8], moreover, because every process is forked from Zygote, we can expect poor runtime performance of ASLR [10].

Another problem of Android security is the customization made by vendors [11]. This aspect is hard to analyze due to the fragmentation of the android hardware and vendors, so we decided to analyze the performance of Android 13 emulated by the Android emulation suite.

2.4.4. Limitations and Improvements

All mentioned research have at least one of the following limitation:

- Lack of a broad OS analysis
- Missing thread execution
- Inadequate sampling size
- Few sections considered
- No multiple allocations considered
- Missing Correlation Entropy estimation
- 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 [1]. As mentioned before the maximum entropy obtainable on the considered system is 35-bit so a k size of $\mathbf{2^{35}}$. To Estimate the entropy using the Shannon formula we will need $\frac{\mathbf{2^{35}}}{\log_{\mathbf{10}}(\mathbf{2^{35}})} = \mathbf{3.261.159.434}$ samples which are way too much to be collected in reasonable time. Even if we consider the best in class, Linux, that in some sections comes close to 30-bit 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, reducing significantly 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 an indicator of the changing bits of the address and only give a rough upper bound to the entropy value.

3 | Approach

Analyzing the performance of ASLR implementation is a task that can be approached in two different ways. The first is to read the actual implementation code inside the kernel of the OS and try to mathematically estimate the entropy of each allocated object based on the mechanism of the randomization algorithm. Walking this path is extremely difficult for various reasons; the main trivial limitation is that we deal with closed-source OS like Windows and MacOS, so finding the source code of the exact implementation we are dealing with is practically impossible. Even if we perform this analysis on Open Source Kernels like Linux, or from the point of view of the manufacturer, there are runtime interactions between memory objects, such as memory fragmentation and allocation patterns, that can significantly change the results. Because of that, the empirical analysis is by far the simpler and most accurate to represent the behavior of ASLR. This approach is the one we have chosen and consists in taking many samples using an ad-hoc script and then performing statistical analysis on the data.

3.1. 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 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.

3.1.1. Memory Objects

Facing the problem of choosing which memory object and sections 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 and the `main()`. Because of this the total number of addresses collected with each sample is around 60 (accounting some platform limitations), thus in the following list we used two placeholders to keep things tight and reduce repetitions.

The placeholders are:

- `@` is used to indicate the flow responsible for the allocation of that object and can be:
  - `M` if it comes from the main flow
  - `ThA` or `ThB` if it was allocated inside one of the two threads
- `SIZE` is used to indicate the size, more on that in the Section 3.1.2
- `#` is used to indicate multiplicity and can be:
  - `1` if is the first allocated for that size and flow
  - `2` if is the second allocated for that size and flow

Must be noted that not every OS has the same characteristics, but we tried to standardize the sampling by using the same nomenclature for objects that are used in the same way across platforms.

Here is the list of each object name with a description:

- `malloc_SIZE_@__#` is the address of an object allocated using the `malloc()` function having `SIZE` as in Section 3.1.2.
- `mmap_single_@__#` is the address of an object of one memory page size allocated using:
  - `mmap()` function with size `4KB` on Linux, Android.
  - `VirtualAlloc()` function with size `4KB` on Windows.
  - `mmap()` function with size `16KB` on MacOS M1.
- `mmap_huge_@__#` is the address of an object allocated using:
  - `mmap()` function with the `MAP_HUGETLB|MAP_HUGE_2MB` flags set on Linux.
  - `VirtualAlloc()` with `MEM_LARGE_PAGES` flag on Windows.
  - MacOS and Android are unable to use Huge Pages.
- `stack_var_@` is the address of a local variable placed on the stack.

- `tls_var_@` is the address of a global variable that is allocated on *Thread Local Storage (TLS)* declared with:
  - `"__thread"` on Linux, MacOS, and Android.
  - `"__declspec(thread)"` on Windows.
- `argv` is the address of the parameters passed to the program when called.
- `env` is the address of the environment variable.
- `global_var` is the address of a global variable.
- `shared_M__#` is the address of a shared memory object mapped using:
  - `MapViewOfFile()` on Windows.
  - `shmat()` on MacOS and Linux.
- `lib__#` is the address of a loaded library.
- `text` is the address of the text area.

We decided to not include some sections in our analysis, mainly regarding not executable pages that are loaded consecutively to other sections so their sampling wouldn't have added more information. In particular, we omitted:

- `.data` and `.bss` on Linux, Android and MacOS.
- `.data, .rdata, .pdata, _RDATA` and `.reloc` on Windows.

as they are always loaded after the .text segment. At the same time `.vsyscall` is omitted because for compatibility reasons on Linux is fixed at `ffffffffff600000` [24].

3.1.2. Allocation Size Choice

It is common for OS to use different types of allocation methods for requests of different sizes in order to optimize performance. For instance, Linux uses a threshold of `128KB` to decide whether to use the legacy system `sbrk()` or the `mmap()` function as an allocation method; moreover, this threshold is variable and is optimized at runtime based on the allocation pattern [22] of the program so we cannot take that for granted. Since the introduction of Folios in Linux 5.18 [23] we have one more variation of allocation method for `mmap()` of sizes `>2MB`, with huge impact on randomization entropy, as we will see in Section 5.1.2

The solution to this problem was to sample in advance a large number of sizes, ranging from `16B` and doubling the size of each allocation to `128MB`. In total, we sampled 24 different sizes, and then using an ad-hoc script we checked the position of each allocation inside the *Virtual Memory Mappings*. By doing so we were able to identify the various threshold for our specific program, OS and allocation strategy exploited by the Kernel. By plotting the results in a table, we then identified a subset of 6 allocation sizes capable of stressing all and each memory segment and allocation technology that we identified, presented in Table 3.1.

	16B	512B	4KB	256KB	4MB	128MB
Linux 6.4	[heap] sbrk()			mmap()	mmap() folio	
Linux 5.17	[heap] sbrk()			mmap()		
MacOS M1 22.04	M_NANO	M_TINY	M_SMALL	M_MEDIUM		M_LARGE
Windows 11	[heap]				NA	NA

Table 3.1: Selected Sizes with allocation segment

We excluded Android from the table, because we have seen that its allocation algorithm, `SCUDO`, allocates every element in a different segment, so we are already oversampling. Anyway, it will sill be useful to highlight any difference in randomization entropy between different sizes.

In conclusion the allocation sizes choosen for the analysis are: `16B`, `512B`, `4KB`, `256KB`, `4MB`, `128MB`.

3.2. Pre-processing and Analysis

The pre-processing phase was guided by the need to reduce the space complexity of the data and also prepare the data to be processed in an efficient way, and we will discuss the implementation choices in Section 4.2.

Regarding the Analysis module, we had to choose which statistics to include in our analysis work. We will discuss them in the following sections.

3.2.1. Probability Distribution

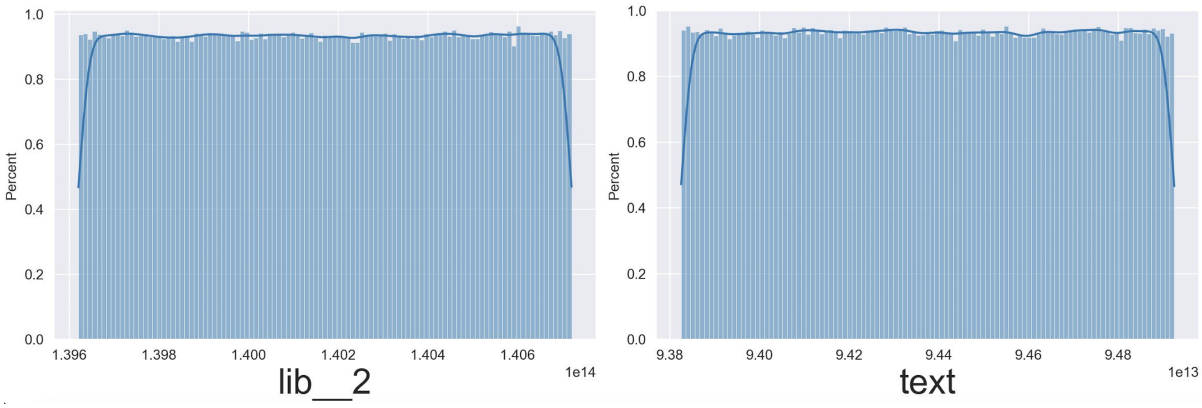

Figure 3.1: Combined KDE and Histogram plot example

The first thing that we wanted to check was the Probability Distribution shape, to do so for discrete distribution we have 2 main tools: *Kernel Density Estimation* and *Binned Histogram*.

The *Kernel Density Estimator (KDE)* is a non-parametric method used to estimate the probability distribution of a dataset, continuous or discrete. It works by placing a symmetric function, typically a Gaussian one, at each data sample position on a plot, and then, combining the contribution of these function graphs, it forms an estimate of the underlying probability density function. When dealing with discrete distribution we have to be careful of the possible misbehavior, mainly related to the smoothing of the final function. For instance, a dataset containing only one value repeated over and over would be plotted as a Gaussian function instead of a vertical bar because of the way the estimation is built.

To overcome this problem we should pair this method with other discrete visualization methods like *Binned Histogram* plot, as shown in Figure 3.1. This plot represents the dis-

crete nature of the dataset but not without challenges, in fact the bin size may have a large impact on the outcome of the analysis. In the end, through trials and errors, we choose to settle with the auto-optimized parameters offered by our graph plotting library, as they were good enough to highlight huge discrepancies from a uniform distribution probability in a qualitative way, as the lack of uniformity will be also reported quantitatively in the Absolute Entropy analysis.

3.2.2. Entropy Estimator

The most important quantitative parameter that we want to analyze is Information Entropy, which is a concept introduced by Shannon [19] in 1948 that measure the information contained in a data source. From another point of view 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 (also called *Plugin Entropy*). Even if in theory it's 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 found is the so-called *Nemenman, Shafee, Bialek (NSB)* estimator [16, 17], which is a coincidence-based estimator that also provides us with *posterior standard deviation* to quantify the uncertainty in the estimation result. Because of this, is still one of the best estimators for under-sampled sources, outperforming both *Shannon Entropy* and *Plugin Entropy* [7]. It has a bias of $\frac{2^{\frac{S}{2}}}{N}$ [16] 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% (0.05) based on the expected final entropy.

In our Absolute Entropy analysis we will encounter, as shown in Section 2.2.1, a maximum

theoretical entropy of 35 bit (S) so we will need to acquire a maximum of:

$0.05 > \frac{2^{\frac{35}{2}}}{N} \Rightarrow N > \frac{2^{\frac{35}{2}}}{0.05} =$ **3,707,276** samples

This method is suitable also for the Correlation Entropy estimation. In the case of Positive Correlation the resulting entropy will be lower than the entropy of the starting section so we will experience a less or equal bias than the absolute one. In the case of Negative Correlation, when the two starting sections are independent, we will see an increase in entropy and so an increment of the bias, if we leave the number of samples unchanged; we can accept this because we are mainly interested in evaluating Positive Correlation entropy, so accurate values for the Negative Correlated sections are not the priority.

3.2.3. Sample Decision

Following the rule presented in Section 3.2.2 we identified the sample lower threshold for each Operating System. To do so, we took several samples and rebooted the OS many times to build an estimate of how many samples we needed to take to obtain a bias lower than 5%. Bear in mind that the resulting bias will be way lower than that, in many objects under 1%, so this is just an upper bound to start with.

Sometimes we used the changing bitmask as an upper bound for entropy, in other cases, when the number would have been too high to be collected in a reasonable time, we refined the estimate using binned Shannon or NSB.

These are the calculated thresholds for the single boot scenario with the methodology used:

Linux 5.17.15 and 6.4.9:

- Max Theoretical Entropy: 35 bits
- Min Samples: 3,800,000

Windows 11:

- Max Theoretical Entropy: 35 bit
- Min Samples: 3,800,000

MacOS M1 Native:

- Max Changing Bitmask: 19 bit
- Min Samples: 15,000

MacOS M1 Rosetta:

- Max Changing Bitmask: 19 bit
- Min Samples: 15,000

Unfortunately, the reboot process is very slow so we cannot take millions of reboot samples, and we have to carefully estimate the numbers. One possible approach is to take a few thousand reboots and then compute a rough estimate on the entropy (**S**) of the interested sections using bitmask.

These are the results we calculated:

Windows 11:

- `global_variable` and `.text` sections: 17-bit changing
- Libraries: 18-bit changing
- Estimated number of reboots to obtain a 5% confidence: 10,000

MacOS ARM M1:

- Libraries: 16-bit changing
- Estimated number of reboots to obtain a 5% confidence: 5,000

MacOS Rosetta M1:

- Libraries: 15-bit changing
- Estimated number of reboots to obtain a 5% confidence: 3,600

Because the sampling process and the rebooting process on Android platform are very slow compared to other operating systems, we decided to recalculate the estimate for the number of samples to be taken also for the single boot scenario. After taking a thousand samples we checked the bitmask for the various sections and the maximum number of changing bits was `23`, inside the .text section. This leaves us with around `58,000` samples to be taken. Another round of tests, refining the results with NSB suggested that `10,000` samples were enough.

For the multiple reboot scenario, we took one hundred samples for one hundred reboots and calculated the bitmask as seen in the previous sections. Unfortunately, it seems that Android changes most of the address bits during reboot, as we found changing bitmask of `33 bit`. Trying to perform enough reboots to cover the possibility of such a large estimated entropy is out of our capabilities as it will take almost `1900000` reboots; spending around `20 seconds` for each reboot it will take us more than a full year. The experience tells us that the final entropy will be way lower than the changing bitmask, so

we decided to go incrementally and stop when we feel satisfied using the posterior standard deviation provided by NSB estimator. In the end, we settled doing 3,000 reboots achieving a bias lower than 1%.

Android 13:

- Max estimated entropy: 16 bit
- Min Samples: 5,000
- Min Reboots: 3,000

4 | Implementation Details

As we can see in Figure 4.1 our analysis tool it's divided into three main components:

- **Sampling Module**: Runs on a remote machine or VMs and takes millions of samples storing them as raw data.
- **Pre-processing Module**: Ingest the raw data generated by the sampling module and convert it to integer format.
- **Analysis Module**: Performs statistical analysis on the data, estimating Absolute Entropy, Correlation Entropy, and providing different statistics.

In the next sections, we are going to present in detail this architecture, describing the choices we made, the challenges we have faced, and how we overcome them.

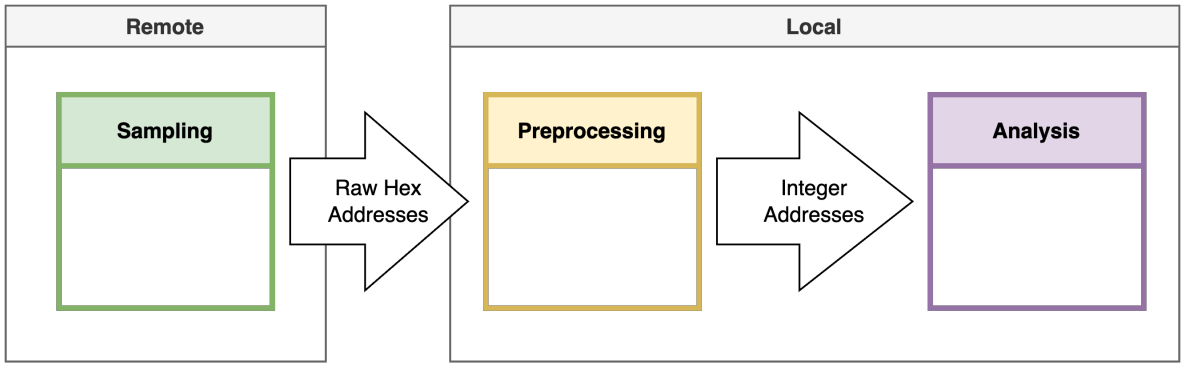

Figure 4.1: Architecture of the ASLR analysing tool

4.1. Sampling

The information we are seeking is the addresses of memory objects and they are usually contained inside the **Virtual Memory Mappings (VMM)**. The VMM for Linux and Android are stored in the form of text files and can be read using the command `cat /proc/PID/maps` where PID is the *process id* of the process we are interested in. On MacOS and Windows, we are forced to use their proprietary tools called *"VMMap"* to

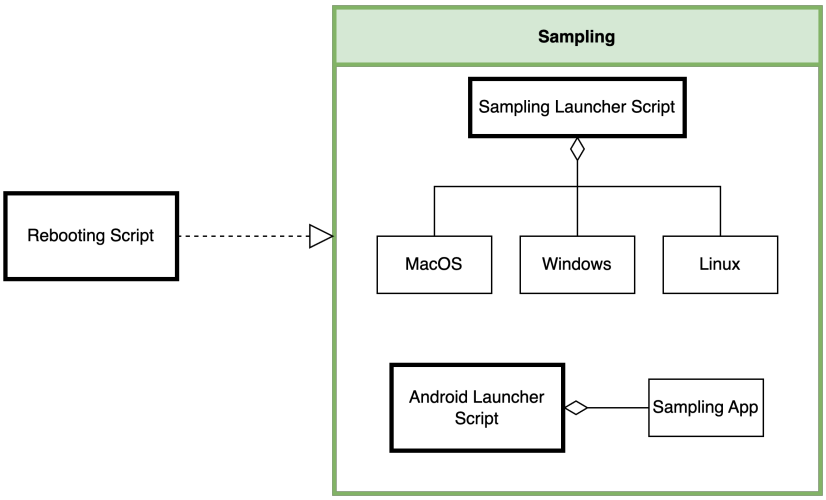

Figure 4.2: Architecture of Sampling Module

access this information. Even if is possible to automate the retrieval process in both cases, so we can use that to build our sampling tool, in doing so we face some serious limitations in terms of sampling speed and information granularity. Every sample, especially on Windows and MacOS, takes a couple of seconds to be retrieved and this is unacceptable when we need millions of them; moreover, the Memory Mappings information does not have the granularity that we need as they report only the page allocation and do not specify the membership of objects, needed to analyze the intra-page entropy of allocations. The solution we have found is to use an ad-hoc tool suite represented in Figure 4.2, the *Sampling Module*. To achieve the desired accuracy in the results of our analysis we calculated that we will need around `3,8 Mln` samples, so reducing the time spent doing each sample is crucial. This module needs to have access to memory at a low level, and it needs to do it fast so we used C language to write the *Sampling Program*. We need to do this sampling process for various operating systems, with less tuning as possible, to avoid rewriting the same code over and over; using C code in the allocation part of our sampling module lets us reuse most of the code on every platform and just adapt the few specific functions that are platform-dependent. During collection and storage, we need easy access to various high-level operations such as file management, directory compression, bash commands, and more; thus, for the *Sampling Launcher*, the Python language was the best choice. This approach is hundreds of times faster than the VMM approach, reaching speeds of around 4000 sample/s on Linux and 300 sample/s on MacOS and Windows.

The last script we need to present is the *Rebooting Script*, which are various bash script that: uses the Sampling Launcher to perform a sampling routine at each reboot and

reboot the operating system autonomously.

Unfortunately, the Sampling for Android cannot be done in the same way, so it was done using an ad-hoc application written in Java that calls a C++ code performing the sample, and then the results were collected using a bash script. We will discuss it in Section 4.1.4.

In the following, we will present in detail each component of this module.

4.1.1. Sampling Program

The sampling program was written in C language for desktop OS and in C++ for Android. The main structure was the same for all programs, changing only the system calls needed to perform various operations like memory allocations and thread launching. It instantiates and allocates many objects in different memory sections, and then prints the addresses of those objects on the standard output to be retrieved by the Sampling Launcher. Because the memory is mainly randomized at the program launch, the sampling program needs only to gather a single sample for each object mentioned in Section 3.1.1 and then exit.

Algorithm 4.1 Sampling Program

```
1:  comm[8]                                          ▷ A support variable
2:  gloabl_var                                       ▷ A global var address
3:  __thread tls_variable                            ▷ A var stored in TLS
4:
5:  procedure MAIN(argc, argv)
6:      tag ← "M"
7:      stack_var_main ← pointer to stack
8:      tls_var ← pointer to TLS
9:      env ← GETENV
10:
11:     shared_mem_1 ← SHMAT(id1)
12:     shared_mem_2 ← SHMAT(id2)
13:
14:     CREATE_THREAD(THREAD_FUNC, 1)
15:     CREATE_THREAD(THREAD_FUNC, 2)
16:
17:     ALLOCATION_FUNC(index, 0, tag)
18:     ALLOCATION_FUNC(index, 1, tag)
19:
20:     print comm and all sampled addresses
21: end procedure
```

The main structure of this program is the one presented in Algorithm 4.1 and is composed

by these main steps:

1. Declaration of three global variables:
    - One array of 8 strings called `comm` to save the addresses from the various allocation.
    - One global variable called `global_var` to obtain the address of a global variable.
    - One global variable declared using `__thread` called `tls_var` to obtain the address of the various Thread Local Storage (TLS).
2. Declaration of a local variable called `stack_var_main` to obtain the main stack address.
3. Recall the current PID, used to double-check the sampling process.
4. Request two shared memory segments (Only Desktop platforms).
5. Two `ALLOCATION_FUNC()` calls as in Algorithm 4.2
6. Recall the address of the environmental variables.
7. Creation of two threads with the function presented in Algorithm 4.3 that each:
    - Declaration of a local variable called `stack_var_th` to obtain the thread stack address.
    - Two `ALLOCATION_FUNC()` as in Algorithm 4.2 .
    - Saving the resulting addresses on `comm` array as a string.
8. Printing of all sampled addresses as a string on the standard output.

The `ALLOCATION_FUNC()` presented in Algorithm 4.2 performs several allocations, following the sizes individuated in Algorithm 3.1.2 and saves by itself the results on `comm` array as a string.

The code needed to be adapted between various OS in particular:

- The included libraries needed to be changed according to OS needs.
- MacOS and Android do not have huge pages to be sampled.
- Android does not have the same shared memory mechanism to be sampled.
- The flag used by the various library functions are different.

Algorithm 4.2 Allocation Function

```
1: function ALLOCATION_FUNC(i, multiplicity, tag)
2:
3:     malloc_16B ← MALLOC(16B)
4:     malloc_512B ← MALLOC(512B)
5:     malloc_4KB ← MALLOC(4KB)
6:     malloc_256KB ← MALLOC(256KB)
7:     malloc_4MB ← MALLOC(4MB)
8:     malloc_128MB ← MALLOC(128MB)
9:
10:    mmap_single ← MMAP(singl_page_size)
11:    mmap_huge ← MMAP(huge_page_size, MAP_HUGETLB|MAP_HUGE_2MB)
12:
13:    index ← (2 * i) + multiplicity − 1
14:
15:    Builds a string of labels and addresses
16:
17:    Save string in comm[index] global var
18: end function
```

Moreover, the Windows implementation of this sampling program required many ad-hoc modifications:

- The `mmap()` function was replaced with the `VirtualAlloc()` function.
- To allocate huge pages, a special token must be requested by the function to the kernel to gain such privileges using an ad-hoc function.
- DLLs must be loaded explicitly to sample libraries' address.
- Various functions' names changed.

The sampling programs are compiled using:

- Linux: `gcc -pthread -w program.c -o program -lm` (we use math libraries to sample the lib position so `-lm` is needed).
- Windows: `cl /Feprogram.exe /Foprogram.obj program.c -w`.
- MacOS: `gcc -pthread -w program.c -o program`.

The easiest way to compile a program to run using Rosetta on MacOS is to run the terminal using Rosetta and then run the compilation procedure inside. Must be noted that on recent versions of `gcc` compiler the programs are compiled by default using the flag `-fPIE` to permit the ASLR system to work and improve security, so it is no more needed. Also cl does not need a custom flag to produce such executable. To double-check

the correctness of these assumptions we verified the header of the executable inside the Sampling Launcher before each run.

Algorithm 4.3 Thread Function

```
1: function THREAD_FUNC(index)
2:
3:     stack_var ← pointer to thread stack
4:     tls_var ← pointer to TLS
5:
6:     Builds a string of labels and addresses
7:     Save string in comm[index] global var
8:
9:     if index == 1 then
10:        tag ← "ThA"
11:    else
12:        tag ← "ThB"
13:    end if
14:
15:    ALLOCATION_FUNC(index, 0, tag)
16:    ALLOCATION_FUNC(index, 1, tag)
17: end function
```

4.1.2. Sampling Launcher

The Sampling Launcher is a set of Python classes that manages the sampling process. The number of threads involved varies between each OS, and so are the pre-sampling checks, thus a series of sub-classes and tuned configurations are used and chosen autonomously when the sampling process is started.

The first thing it does is run checks to assure that the environment is set correctly:

1. Recompiles the executable of the sampling program as described in Section 4.1.1 to assure we are using the up-to-date version.
2. Checks the headers of the just compiled executable to verify that it is compiled for the correct architecture we are targeting (especially important on MacOS).
3. Checks the headers of the just compiled executable to confirm that it is indeed a Position-Independent Executable (PIE):
   - Linux: the presence of **"pie"** string in the output of `file` command.
   - Windows: the presence of **"Dynamic base"** and **"High Entropy Virtual Addresses"** strings in the output of `dumpbin /headers` command.

- MacOS: the presence of **"pie"** string in the output of `otool -hv` command.

4. Enables huge pages with `sudo -S sysctl -w vm.nr_hugepages=512` (Linux only).
5. Checks that Address Space Layout Randomization (ASLR) is active with the maximum entropy available.

The feasibility of verifying the status of Address Space Layout Randomization (ASLR) varies across different operating systems due to limitations. On Linux, ASLR status can be assessed by examining the output of the command `cat /proc/sys/kernel/randomize_va_space`. A value of 0 indicates that ASLR is disabled, a value of 1 signifies *Conservative Randomization* wherein `brk()` memory regions remain unrandomized, and a value of 2 corresponds to *Full Randomization*. For Windows 11, a direct system-wide ASLR status check is not available, therefore, a manual inspection of the *Exploit Prevention* settings within the control panel is necessary. It is crucial to ensure that both Bottom-up ASLR and High-Entropy ASLR settings are activated. In contrast, MacOS does not provide an option to modify the default ASLR setting (enabled); moreover, direct verification of this setting is not feasible within MacOS.

After these checks, combined with the checks regarding the executable headers we can be comfortable starting the sampling process spawning various threads to increase the sampling speed. We noticed that disabling *Real-Time Threat Protection* on Windows 11 doubled the sampling speed, so we turned that off during the sampling process. The tool provides a CLI to monitor the progress, speeds, and also an estimated time of finish.

The samples are collected directly from the standard output of the Sampling program and then saved on different text files after a buffer is filled, one for each thread, to avoid concurrency issues and to reduce the number of writes to storage. At the end of the sampling process the files are compressed into a zip archive and the checksum of the archive is compiled and saved along with other metadata in a text file, ready to be transferred to the pre-processing module.

4.1.3. Rebooting Script

As mentioned in Section 2.1.1, MacOS, Windows 11, and Android randomize some sections only at system startup. Even if for a local attacker this is a huge advantage, as every process shares the same addresses, in a distributed attack scenario when we can target multiple devices at once, the study of the entropy of those sections is very relevant. To do so the only way is to reboot our device to force the randomization of sections and then take a certain amount of sample each reboot.

Even if the rebooting script is customized for each Operating System the main structure is the same for both Windows 11 and MacOS and is the following:

1. Write the number of reboots we want to perform to a file called `counter.txt`.
2. Set the number of samples we want to take at each reboot.
3. Start the sampling using the Sampling Launcher.
4. Decrease the counter, write the updated value to `counter.txt`, and then reboot.

In Algorithm 4.4 we described the pseudo-code of this script. We add the script as a startup application so that the process could be automated; to do so, we have to remove any disk protection and user password, so that the system can be restarted without human intervention.

For what concerns Android the approach is a little different and we are going to discuss it in Section 4.1.4

Algorithm 4.4 Rebooting Script

```
1: procedure
2:
3:     SAMPLE ← samples at reboot
4:     COUNT ← counter.txt                          ▷ reboots to perform
5:     STOP ← 0                                     ▷ when we want to stop
6:
7:     if COUNT < STOP then
8:         OUTPUT("SamplingFinished")
9:         EXIT
10:    end if
11:
12:    SAMPLING_LAUNCHER.PY(SAMPLE, COUNT)          ▷ starts the sampling
13:    counter.txt ← COUNT                          ▷ reboots to perform
14:    REBOOT
15:
16: end procedure
```

4.1.4. Android

To analyze ASLR performance on Android we cannot use the same approach seen with other Operating Systems as it runs only APK applications written in Java or Kotlin. Our Sample Launcher is no longer useful, as it runs in Python, so we have to develop a new approach from scratch to extract information from the device and save them. Using Android Studio we have written a simple Java App that loads a library, containing the

same Sampling Program seen in Algorithm 4.1 slightly adapted and written in C++. In particular, two crucial customization were made:

- To extract the information we can no longer use the standard output, but we have to rely on the logging functionality `__android_log_print()`, as it is accessible from outside the device using `ADB`.
- To restart the application to collect samples without rebooting the device, we added an `exit(1)` at the end of the code to inject a failure and make the application restart way faster than doing it manually using `ADB`.

Algorithm 4.5 Android Reboot Launcher

```
1:  procedure
2:
3:      SAMPLES ← samples at reboot
4:      REBOOT ← reboots to perform
5:      DEVICES ← n.virtual devices
6:
7:      for DEVICES do
8:          EMULATOR(@Device_1, read − only, cold − boot)
9:      end for
10:
11:     while not all devices are online do
12:         SLEEP(10)
13:     end while
14:
15:     R_DEV ← REBOOT/DEVICES          ▷ number of reboot for each device
16:
17:     for each running device do
18:         DEVICE_HANDLER(device_name, R_DEV, SAMPLES)
19:     end for
20:
21:
22: end procedure
```

This application can now be compiled and loaded inside the chosen device. Even though we could use a hardware device and then control the process using ADB commands, the integration of Android Emulator inside Android Studio made the emulation path the best choice, both for reproducibility and simplicity. Android Emulator (AE) is software that emulates an Android device using the selected Android version and provides various customization. The one that is relevant for our research is the ability to perform a hot-start, enabled by default, using a previous ram image to reduce startup time. Unfortunately, this feature impact negatively

on our research as we want the device to renew the memory layout every time we restart it, so we have to make sure is disabled; as a consequence, the reboot now is way slower, but it randomizes memory as we expect from an actual hardware device. AE is a convenient choice also because it is fully usable inside the terminal in headless mode, so we can control both the virtual device and the logging procedure using bash scripts.

We built two scripts: *Android Reboot Launcher* (Algorithm 4.5) and *Device Handler* (Algorithm 4.6). The first script launches as many Android emulator devices as needed, in headless mode and without using a snapshot (Cold-boot); then it waits for each device to come online and it calls another script, the Device Handler, passing to it the device name, the number of samples requested and the number of reboots needed for that specific device. Using multiple devices, in theory, permits us to parallelize a bit this very slow process of rebooting, in practice, our machine could only handle one device at a time.

Algorithm 4.6 Device Handler

```
1: procedure DEVICE_HANDLER("DEV_NAME, REBOOT, SAMPLES")
2:
3:     R_COUNTER ← 0
4:     S_DELAY ← 5
5:     R_DELAY ← 15
6:
7:
8:     while R_COUNTER < REBOOT do
9:         while N_SAMPLES < SAMPLES do
10:            LOGCAT(DEV_NAME)
11:            N_SAMPLES ← samplecounts
12:        end while
13:        REBOOT(DEV_NAME)
14:        R_COUNTER ← R_COUNTER + 1
15:        SLEEP(10)
16:    end while
17:
18:    KILL(DEV_NAME)
19:
20: end procedure
```

The Device Handler, received the device name, starts communicating with the virtual device using ADB and it execute this procedure:

1. Launches the Sampling application on the device.

2. Listen for the data stream coming from the device using `logcat`.
3. Save the stream to a text file and count the number of samples collected.
4. When the number of samples is enough, reboot the device using `adb reboot`.
5. When the number of reboots is enough, shut down the device and exit.

Clearly, we can use this tool also for sampling one single reboot, as we can request 1 reboot and as many samples as we want.

4.2. Preprocessing

The role of the preprocessing module is to convert and build an efficient and fast data frame needed in the analysis phase. It starts from the text output of the sampling program and collected by the sampling launcher.

This was done using a Python script that:

1. Extract and verify the archives coming from the remote machine.
2. Parse the content of the text files.
3. Build a `.csv` file with hexadecimal data.
4. Convert `.csv` file into `.parquet` integer data.

The first step is to extract and verify the checksum of all archives transferred from the remote machines that performed the sampling precess. Because the samples are strings with many duplicated characters (for instance, the labels and the fixed bytes of the addresses are repeated), the compression process is very efficient in keeping the dataframe small and easily transferable; moreover, it permits us to easily check the integrity of the data after the transmission, using a MD5 checksum: this was done to rule out any outlier values due to file corruption.

At this point, we are left with `N` text file, one for each thread used in the sampling module. This long text file, represented in Figure 4.3, contains a row for each sampled object, and each row is composed of a label and an hexadecimal address. Each sample run is intercalated by a special character, `#`, and a number used to distinguish between them. To parse the content we use a function that opens each text file one by one, reads the content, and builds a dictionary with the addresses; then, when we encounter a `#` char we output the samples as a single long csv row, where each column represents a memory object.

```
#0
malloc_16B_M__1 0x600001b2c010
malloc_512B_M__1 0x7f78ef004240
malloc_4KB_M__1 0x7f78ef808200
malloc_256KB_M__1 0x7f78f0008000

                    ...

lib__1 0x7ff80bcb9f25
lib__2 0x7ff80bda5001
text 0x100a01930
PID 775
#1
malloc_16B_M__1 0x6000021f8040

                    ...
```

Figure 4.3: Example of collected text file

At the end of the parsing, we obtain a file called `raw_data.csv`, in Figure 4.4, containing all sampled data in hexadecimal format, where every run belongs to a different line and every object to a different column. The first part of the preprocessing is finished, now we need to make the data frame more performant.

As mentioned before the data we are dealing with contains a lot of repeated characters and the raw file we obtained is large and slow to use; moreover, the pieces of information are stored as strings, while our analysis deals with them as integers, so the `.csv` format isn't the best candidate for our purpose. We found a good alternative in Parquet file, as it stores data in their native format and also reduces the sizes on disks due to default column data compression. The next part takes the `raw_data.csv` and after parsing the hexadecimal addresses into a 64 bit long integer, writes them to disk in `.parquet` format; this conversion is an obliged step, because neither Pandas nor Polars (the libraries used to analyze the data) support natively hexadecimal numbers.
In the end, we obtained a fast and light data frame ready to be used.

4.3. Analysis

The Analysis process is both graphical and numerical, and we also need to control the flow of the analysis to account for platform peculiarities. To achieve all these objectives the best suited tool is *Jupyter Notebooks* with Python programming. This also provides us with a huge choice in terms of data science libraries to perform our tasks.

```
reboot,malloc_16B_M__1,malloc_512B_M__1,...
0,600003020040,7fb7a67046d0,7fb7a6808800,...
0,6000016ac010,7fed7f004080,7fed7f808200,...
0,60000117c010,7f9184004080,7f9184808200,...
0,600002368040,7fac6cf046d0,7fac6d008800,...
0,60000193c040,7f8217f046d0,7f8218008800,...
0,600000694040,7f96647046d0,7f9664808800,...
0,6000013a8040,7fd9377046d0,7fd937808800,...
0,600000414040,7fa687f046d0,7fa688808200,...
0,600000304020,7f7d30f04290,7f7d32008200,...
0,600003578040,7f915ef046d0,7f915f008800,...
...
```

Figure 4.4: Example of `raw_data.csv`

To isolate each step we built 6 different *Jupyter Notebooks* :

1. Range Calculation.
2. Memory Layout.
3. Probability Distribution.
4. Absolute Entropy Estimation.
5. Difference Calculation.
6. Correlation Entropy Estimation.

During the research phase also others were used but were not introduced in the final analysis suite

For the graphical and data visualization portion of the analysis, we decided to use the library *Seaborn* as it is a well-documented and broadly used tool based on the famous *Matplotlib*. It joins the simplicity of pre-build templates and graphing methods, without compromising on functionality and flexibility thanks to its Matplotlib backbone. We used this library for the Probability Distribution Plot and the Memory Layout. The first is a self-adapting-size subplot, composed of several histograms drawn using the `histplot` function, one for each memory object sampled. The second is a customized `scatterplot`, usually used to visualize the distribution of a group of points in a 2-dimensional space. In our case, we wanted to draw a figure representing the position of the various objects inside the memory so we were interested in a single dimension; however, plotting all 60 objects on a single line would be chaotic, so using the `scatterplot` we spaced them on the Y ax while having the addresses on the X ax.

The management of data frames (loading, manipulation, and storing) was mainly done in *Pandas*. This library is one of the most used data scientist tools thanks to its community support and exhaustive documentation. The only drawback we found is the slow performance when dealing with very large datasets. In those rare cases, that appeared only during the research phase and were excluded in the final analysis pipeline, we used *Polars*, a performance-optimized data science library that permits streaming of the data without loading the entire set in memory.

To perform the entropy estimation, as mentioned in Section 3.2.2, we had various choices in terms of algorithms; the final choice was the *NSB algorithm*. Lucky for us the algorithm was already implemented in Python language in the open source library **ndd** by *Simone Marsili* [15]. This tool was used both in the estimation of Absolute Entropy and Correlation Entropy. To use it we needed to first calculate the number of addresses with non-zero probability of being used by ASLR. We used the difference between the highest and lowest address of each object, to identify the range. In theory, the range isn't the real number of possible addresses as we didn't account for the fixed page offset, however, this aspect was revealed to be of minor importance during testing; moreover, the page offset is one of the main problems when we consider ASLR performance so including that in the range of possibility was considered a fair approximation.

To calculate the Correlation Entropy we need to consider the concept of distance inside memory, better known as offset. To calculate the distance between every object to each other we simply performed a column-wise subtraction. This process resulted in one data frame for each object considered, in our case around 60, each one the size of the original in terms of column numbers and rows; also, disk space occupation for each one was comparable with the original. In our Linux analysis, we collected around `3.8Mln` samples, that in their converted and compressed form inside a parquet file occupied `1.3GB` of disk space. Storage and manipulation of 60 data frames of this size, for each system that we considered, was revealed to be a challenging task.

To reduce the size of those files we needed to observe the nature of the offset itself. The entropy operates on the absolute value, so, even tho the subtraction isn't a commutative operation, the results for positive and negative offset are the same; because of this, we could cut in half the time and space required by this analysis, considering only one of the two directions of the distance between objects, and cloning the results for the other direction; this provided us with a zero-diagonal symmetrical matrix containing the Correlation Entropy results.

5 | Experimental Evaluation

In this section, we are going to present the tested configuration and briefly discuss the strength and weaknesses of each system in the following categories:

- Probability Distribution
- Memory Layout
- Absolute Entropy
- Correlation Entropy

During the analysis process, we took **20 bit** of entropy as the threshold of reference. We considered good results everything above this value and bad results everything under. This threshold is artificial, as in reality there is no real value of "safeness", and it strongly depends on how many `tries per second (tps)` an attacker is able to perform. As this was the value used also in other research we decided to stick with it [13, 14]. Unfortunately, there are no real metrics on the tries per second that a system can handle, as it depends on many variables that are impossible to track, however, during this research we experienced speeds ranging from 30 to 500 samples per second locally with consumer-grade hardware. Remotely, is even more difficult to estimate as depends on the size of the exploit and the network speed considered[12]. For reference, an attacker performing around `300 tps` would take on average 1 hour to successfully brute force a memory object with 20 bits of entropy. The same attacker would be able to break the randomization of 16.5 bit of entropy in less than a minute.

To present Absolute Entropy results we grouped the objects in categories:

- **executable:** `text, lib__1, lib__2`;
- **stack:** `stack_var_ThA, stack_var_ThB, stack_var_M`;
- **mmap():** all object allocated using `mmap()` or `VirtAlloc()` without using huge pages;
- **main malloc():** all object allocated using `malloc()` from the `main()`;

- **thread malloc():** all object allocated using `malloc()` from thread `ThA` and `ThB`;
- **1st malloc():** the first allocated object using `malloc()` from thread `ThA`, `ThB` and `main()`;
- **2nd malloc():** the second allocated object using `malloc()` from thread `ThA`, `ThB` and `main()`;
- **mmap() huge:** all object allocated using `mmap()` or `VirtAlloc()` with huge pages;
- **16B:** all object allocated using `malloc(16B)`;
- **512B:** all object allocated using `malloc(512B)`;
- **4KB:** all object allocated using `malloc(4KB)`;
- **256KB:** all object allocated using `malloc(256KB)`;
- **4MB:** all object allocated using `malloc(4MB)`;
- **128MB:** all object allocated using `malloc(128MB)`;
- **shared:** `shared_M__1, shared_M__2`;
- **other:** `argv, env, global_var, tls_var_ThA, tls_var_ThB, tls_var_M`;

For each group, we show the minimum (**MIN**) and maximum entropy (**MAX**), useful for identifying the low-performance objects and the best-performing ones. The minimum is by far the most relevant information, as the object having minimum entropy will be the preferred target in case of an attack. On the other hand, the maximum gives an upper bound to group performance, and a very low maximum entropy may be a sign of a broken ASLR implementation. In addition, we calculated the average entropy (**AVG**) to better frame the overall behavior of the group. The group's division has been made to cover most of the possible categorizations considered, so you can match the information of different groups to spot weaknesses.

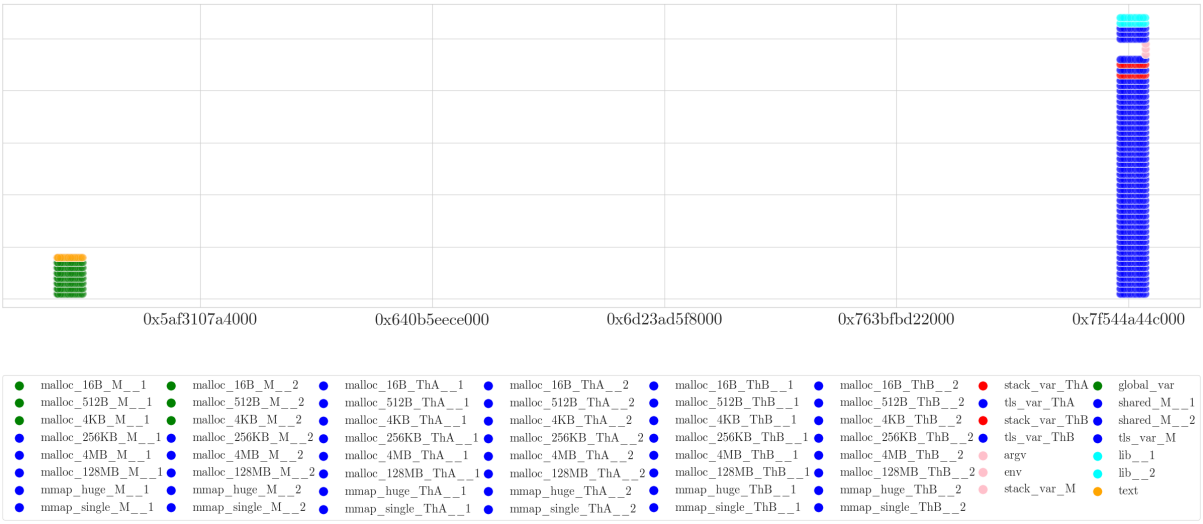

Figure 5.1: Memory Layout Linux 5.17.15

Hypervisor	`QEMU 7.1.0`
CPU	`i7-4790K`
RAM	`32GB RAM`

Table 5.1: General Configuration Linux

5.1. Linux

All Linux sampling was executed running the OS inside an emulated environment with hardware and software configuration as in Table 5.1. The specific sampling configuration for kernel versions 5.17.15 and 6.4.9 are indicated respectively in Table 5.2 and Table 5.5. Linux kernel does not randomize objects at boot but every time a process is launched, the samples are collected in a single boot instance.

5.1.1. Linux 5.17.15

Memory Layout. As we can se in Figure 5.1 the memory layout is mainly divided into 2 sections with a large empty section in the middle: on the left, starting with lower addresses, we have the section used by the kernel to store allocations under 256KB, such as those positioned in the heap using `sbrk()`, and the text segment; on the right of the image, positioned among high memory addresses, we have the `mmap()` segments where libraries and allocations greater than 256KB are located. Additionally, the main stack also belongs to this section, though its span is relatively minor compared to others. It's important to note that while addresses may be closer together, this is not necessarily an

OS	Ubuntu 23.04
Architecture	x86_64
Kernel	Linux 5.17.15
Lib	GLibC 2.37
First Boot Samples	3,800,000

Table 5.2: Information Linux 5.17.15 Sampling

	MIN	MAX	AVG		MIN	MAX	AVG
executable	27.41	27.41	27.41	16B	14.74	27.41	19.03
stack	19.02	27.40	21.81	512B	14.74	27.41	19.03
mmap()	27.41	27.41	27.41	4KB	14.74	27.41	19.03
main malloc()	27.41	27.41	27.41	256KB	23.72	27.41	25.04
thread malloc()	14.74	24.77	17.39	4MB	18.93	27.41	20.76
1st malloc()	14.74	27.41	20.78	128MB	15.66	27.41	18.66
2nd malloc()	14.74	27.41	19.73	shared	27.41	27.41	27.41
mmap() huge	18.71	19.02	18.87	other	19.02	27.41	23.79

Table 5.3: Entropy Groups Linux 5.17.15

indication of low entropy, as it depends on the density. However this grouping suggest high presence of correlation between objects in particular the ones allocated using `mmap()`.

Probability Distribution. From Figure A.1 the probability distribution It appear uniform, without raising significant concerns under in this aspect. The graph represents the desired randomization distribution across all memory sections.

Absolute Entropy. Looking at Table 5.3 we see that the entropy of stack objects varies from a nearly acceptable 19 bit to an excellent 27.4 bit; further inspection at complete data in Table A.1 tells that the best performing one is the main stack while the low entropy one belongs to threads. We know from the documentation of `pthread_create()` that the default size of the thread stack is 2MB [9]; 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. All executable objects, `text`, `lib__1` and `lib__2` perform greatly with over 27 bit of entropy. In general, we can observe a difference between objects allocated by the main program flow and the ones allocated by threads. Regarding the main flow, the first round of allocations experiences an excellent **27.4 bits** of entropy, except for the `mmap()` using huge pages, which stops at around **19.6 bits**; this reduction is expected as huge pages use larger page-offset: 12 bit for 4KB pages versus 21 bit for 2MB huge pages. A reduction in entropy is clearly visible during the second round of allocations greater than 4 MB, dropping as low as 15.66 bits; this highlights how the allocation pattern significantly affects the performance of ASLR, as the available

path	ent	diff	path	ent	diff
lib_1←lib_2	0.00	-27.41	lib←malloc_512B_Th	15.15	-12.25
lib←tls_var_M	0.00	-27.41	lib←malloc_16B_Th	15.15	-12.25
lib←shared	0.00	-27.41	lib←mmap_single_M	0.00	-27.41
lib←tls_var_Th	9.00	-18.41	lib←mmap_huge_M	9.00	-18.41
lib←stack_var_Th	9.00	-18.41	lib←malloc_128MB_M	0.00	-27.41
lib←mmap_single_Th	0.98	-26.43	lib←malloc_4MB_M	0.00	-27.41
lib←mmap_huge_Th	11.54	-15.87	lib←malloc_256KB_M	0.00	-27.41
lib←malloc_128MB_Th	14.72	-12.69	text←global_var	0.00	-27.41
lib←malloc_4MB_Th	11.17	-16.24	text←malloc_4KB_M	13.00	-14.41
lib←malloc_256KB_Th	4.53	-22.88	text←malloc_512B_M	13.00	-14.41
lib←malloc_4KB_Th	15.15	-12.25	text←malloc_16B_M	13.00	-14.41

Table 5.4: Positive Correlation executable path Linux 5.17.15

OS	Ubuntu 23.04
Architecture	x86_64
Kernel	Linux 6.4.9
Lib	GLibC 2.37
First Boot Samples	3,800,000

Table 5.5: Information Linux 6.4.9 Sampling

space decreases, requiring `mmap()` to work harder to find empty spots, thus decreasing the absolute entropy. For threads instead, nearly all allocations have an entropy of 15 bits, while the stack and *TLS (Thread Local Storage)* perform better with around 19 bits. Overall the executable objects are well-randomized while the most weak objects are the ones belonging to threads.

Correlation Entropy. As we expected from the memory layout we can identify many objects that have Positive Correlation with other objects; this behavior is clearly evident in Figure A.2. Raising attention, we can see the executable objects lowering their entropy by almost 14 bits, or in other words, a leak can reduce the attack effort by 16,000 times. The Positive Correlation path raising warnings are presented in Table 5.4. Moreover, all mapped area suffers from consecutive allocation.

5.1.2. Linux 6.4.9

We can see comparing Figure 5.2 and Figure 5.1 that the performance of the Linux kernel in terms of memory layout is very consistent between the versions considered; the same is true for the probability distribution as we can see in Figure A.3, so you can refer to Section 5.1.1 for comments regarding these aspects.

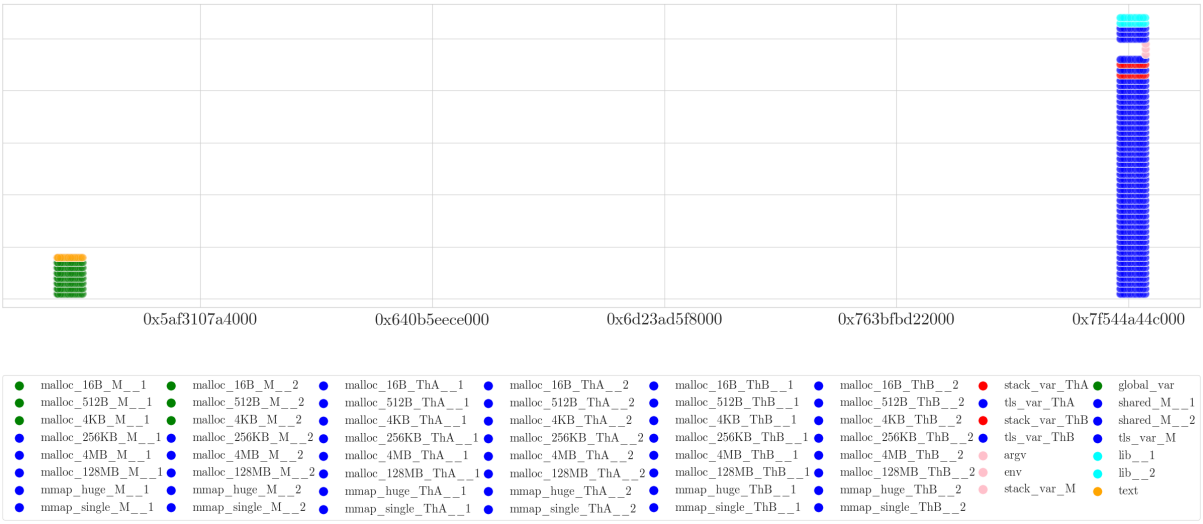

Figure 5.2: Memory Layout Linux 6.4.9

	MIN	MAX	AVG		MIN	MAX	AVG
executable	19.02	27.41	24.61	16B	14.93	27.41	19.11
stack	19.02	27.40	21.82	512B	14.93	27.41	19.11
mmap()	27.41	27.41	27.41	4KB	14.93	27.41	19.11
main malloc()	19.61	27.41	24.55	256KB	23.50	25.83	24.29
thread malloc()	14.93	24.01	17.44	4MB	19.02	20.20	19.51
1st malloc()	14.93	27.41	19.86	128MB	16.07	19.61	17.38
2nd malloc()	14.93	27.41	19.64	shared	27.41	27.41	27.41
mmap() huge	18.80	19.02	18.90	other	19.02	27.41	23.79

Table 5.6: Entropy Groups Linux 6.4.9

Absolute Entropy. Where a difference shines is in the Absolute Entropy, in fact we can see in Table 5.7 that now all objects greater than 2MB experience a huge reduction in entropy; from Table A.2 we confirm the same reduction for `lib__1`. Further investigation highlighted that this behavior started in version `5.18` of the Linux Kernel and is still present in the last available version at the moment, `6.4.9`, and so we choose that for our analysis.

We tracked back the changes and found that a new memory management system is being introduced in the Linux Kernel, with the implementation of Folios. This work started in version `5.14` but only in version `5.18` did they activate Large Folios, so we suspect that they may be responsible. Linux Folio is a memory structure introduced to increase performance and reduce memory fragmentation, as it groups multiple consecutive memory pages of 4KB, being now represented 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

path	ent	diff	path	ent	diff
lib_2←lib_1	9.00	-18.41	lib_1←shared	9.00	-10.02
lib_2←tls_var_M	0.04	-27.37	lib_1←tls_var_Th	0.00	-19.02
lib_2←shared	0.00	-27.41	lib_1←stack_var_Th	0.00	-19.02
lib_2←tls_var_Th	9.00	-18.41	lib_1←mmap_single_Th	9.00	-10.02
lib_2←stack_var_Th	9.00	-18.41	lib_1←mmap_huge_Th	2.76	-16.26
lib_2←mmap_single_Th	1.17	-26.24	lib_1←malloc_128MB_Th	6.72	-12.30
lib_2←mmap_huge_Th	11.76	-15.65	lib_1←malloc_4MB_Th	2.32	-16.70
lib_2←malloc_128MB_Th	15.24	-12.17	lib_1←malloc_256KB_Th	6.23	-12.79
lib_2←malloc_4MB_Th	11.20	-16.21	lib_1←malloc_4KB_Th	6.38	-12.65
lib_2←malloc_256KB_Th	5.16	-22.25	lib_1←malloc_512B_Th	6.38	-12.65
lib_2←malloc_4KB_Th	15.38	-12.03	lib_1←malloc_16B_Th	6.38	-12.65
lib_2←malloc_512B_Th	15.38	-12.03	lib_1←mmap_single_M	9.00	-10.02
lib_2←malloc_16B_Th	15.38	-12.03	lib_1←mmap_huge_M	0.00	-19.02
lib_2←mmap_single_M	0.00	-27.41	lib_1←malloc_128MB_M	0.59	-18.44
lib_2←mmap_huge_M	9.00	-18.41	lib_1←malloc_4MB_M	0.59	-18.44
lib_2←malloc_128MB_M	8.74	-18.67	lib_1←malloc_256KB_M	8.23	-10.79
lib_2←malloc_4MB_M	8.74	-18.67	text←global_var	0.00	-27.41
lib_2←malloc_256KB_M	1.35	-26.06	text←malloc_4KB_M	13.00	-14.41
lib_1←lib_2	9.00	-10.02	text←malloc_512B_M	13.00	-14.41
lib_1←tls_var_M	9.00	-10.03	text←malloc_16B_M	13.00	-14.41

Table 5.7: Positive Correlation executable path Linux 6.4.9

not fixed. Its size is a power of two and it is aligned with its size [23], 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 9 bit reduction in entropy, and our results in Table A.3 confirm that. In fact, the sampled lib experiencing the reduction is sampled from the `libc` that is indeed bigger than 2MB. This is a huge reduction for an executable memory section and gives an attacker almost `400x` more chances of success compared to Linux version `5.17.15`.

Correlation Entropy. Again, the Correlation Entropy matrix in Figure A.4 is very similar to the Linux 5.17 one in Figure A.2, however, new Positive Correlation paths are present compared to Linux 5.17.15 due to libraries not being fully correlated as before. The updated paths are indicated in Table 5.7.

5.2. MacOS

MacOS sampling was performed on a Macbook Pro M1 2020 configured as in Table 5.8. Being MacOS a system that randomizes executable objects only at boot, we rebooted the machine several times to collect the samples. We sampled both Ma Precise configuration

OS	`MacOS Ventura 13.4.1`
Kernel	`Darwin 22.5.0`
CPU	`ARM M1`
RAM	`8GB Unified`
Hardware Arch.	`arm64`

Table 5.8: General Configuration MacOS

Architecture	`arm64`
First Boot Samples	`1,000,000`
Reboots	`5,000`
Reboot Samples	`500`
Total Reboot Samples	`2,500,000`

Table 5.9: Information MacOS M1 Native Sampling

for each sample run can be found in Table 5.9 and Table 5.12.

5.2.1. MacOS M1 Native

Memory Layout. The memory on MacOS using Native ARM architecture, visible in Figure 5.3, is 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 suggest high correlation between objects.

Probability Distribution. As we can see from the graphs in Figure B.1, there are some uniform allocation regarding the `TLS` objects and `MALLOC_NANO` objects. The others have a distributions characterized by high spikes or large groups, indicating a very low entropy. On some section, the randomization is probably linked to the intrinsic not deterministic positioning of allocation and not to explicit ASLR action. Moreover, the library's position is fixed.

Absolute Entropy. As emerge from Table 5.10 the overall the entropy is low, with poor randomization on executable objects: libraries are fixed, and text have insufficient randomization entropy (11.5 bit). As mentioned before, the objects allocated with `malloc()` are basically not randomized, with just a little bit of entropy due to uncertainty in the optimization phase of the allocation. `mmap()` does better with around 12.5 bit, but still insufficient. The complete results of Absolute Entropy analysis are provided in Table B.1.

Correlation Entropy. In the correlation matrix presented in Figure B.2 we can see an unusual behavior where the security in terms of entropy increases for almost every object,

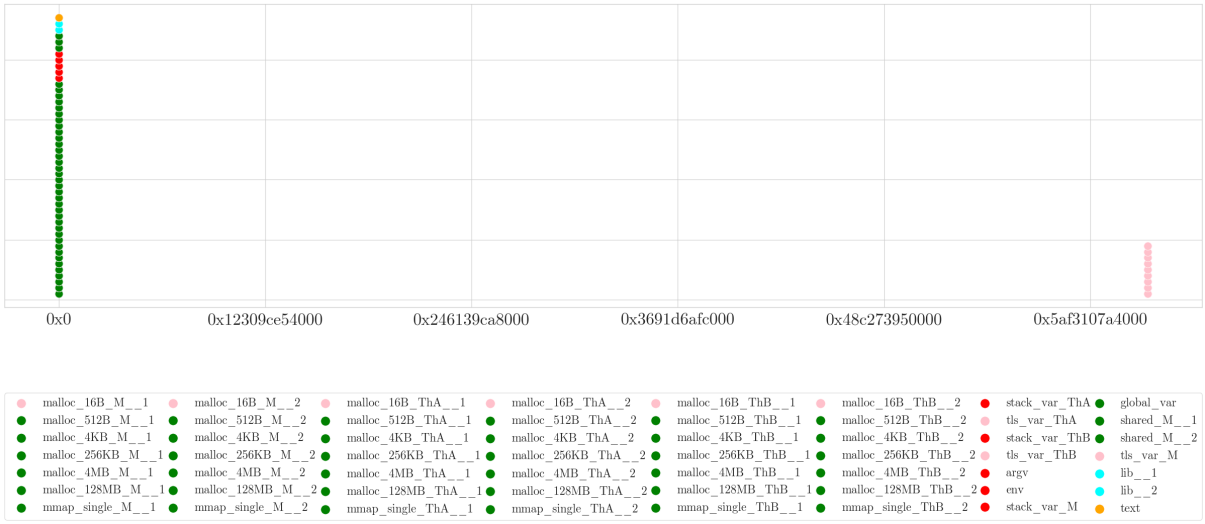

Figure 5.3: Memory Layout MacOS M1 Native

	MIN	MAX	AVG		MIN	MAX	AVG
executable	0.00	11.58	3.86	16B	12.58	14.55	13.68
stack	11.58	11.82	11.66	512B	7.55	9.98	8.95
mmap()	11.59	12.61	12.29	4KB	7.46	9.18	8.42
main malloc()	3.19	12.58	6.85	256KB	3.19	5.35	4.40
thread malloc()	4.55	14.55	8.34	4MB	3.23	5.36	4.56
1st malloc()	3.19	14.05	7.72	128MB	6.65	7.86	7.25
2nd malloc()	3.27	14.55	8.03	shared	11.58	11.58	11.58
mmap() huge	NA	NA	NA	other	11.58	13.52	12.62

Table 5.10: Entropy Groups MacOS M1 Native

this is because this correlation is the result of two uncorrelated random variables, thus increasing the resulting entropy. We call this Negative entropy. However, this is irrelevant when the starting regions have low to none randomization, as it is easier to brute force directly the target object. Some objects are consecutive to other executable objects, as we can see from Table 5.11.

Reboot Changes. After reboot only the entropy of the libraries, previously fixed in position, is affected, with 12.2 bit of absolute entropy. The other objects are unchanged as we can see in Table B.2. The Probability Distribution of the newly randomized sections presented in Figure B.3 appears uniform. In the end, the randomization process performed during reboot is still unable to protect against a brute-force attack due to its low entropy.

path	ent	diff	path	ent	diff
text ← shared	0.00	-11.58	text ← mmap_single_Th	2.16	-9.42
text ← global_var	0.00	-11.58	text ← mmap_single_M	0.01	-11.58

Table 5.11: Positive Correlation executable path MacOS M1 Native

Architecture	arm64
Emulator	Rosetta
Emulated Arch	x86_64
First Boot Samples	1,000,000
Reboots	5,000
Reboot Samples	500
Total Reboot Samples	2,500,000

Table 5.12: Information MacOS M1 Rosetta Sampling

5.2.2. MacOS M1 Rosetta

Memory Layout. In Figure 5.4 we can see that the memory layout on MacOS using Rosetta is divided into three section. The lower one locates explicitly mapped pages, stack of main and threads and the text. The middle one correspond to the higher of the Native Implementation, accommodating `MALLOC_NANO` and `TLS` variables. The higher one is dedicated to `malloc()` objects and libraries that this time presents more visual distribution, so we can expect higher entropy compared to the Native implementation.

Probability Distribution. Clearly visible in Figure B.4, Rosetta does a good job distributing the allocations, with all `malloc()` objects having a uniform distribution. The remaining objects have a uniform distribution among three distinct groups while the libraries as we expected are fixed in memory.

Absolute Entropy. As is clearly evident from Table 5.13 the absolute entropy is low. As we can see in Table B.3, executable objects are poorly randomized: libraries are fixed and text is randomized with just 13.5 bit. However, it does a better job compared to Native implementation, mainly in the `malloc()` objects that now are randomized with entropy ranging from 12 to 19 bit. Overall the randomization is insufficient.

Correlation Entropy. In Figure B.5 we observe a lot of memory regions with Negative Correlation, in particular the `malloc()` objects appear to be completely uncorrelated with other regions. In terms of the relevant Positive Correlation path, we can see a complete correlation between text objects and mapped pages. Other Positive Correlation Path that lead to executable objects are reported in Table 5.14.

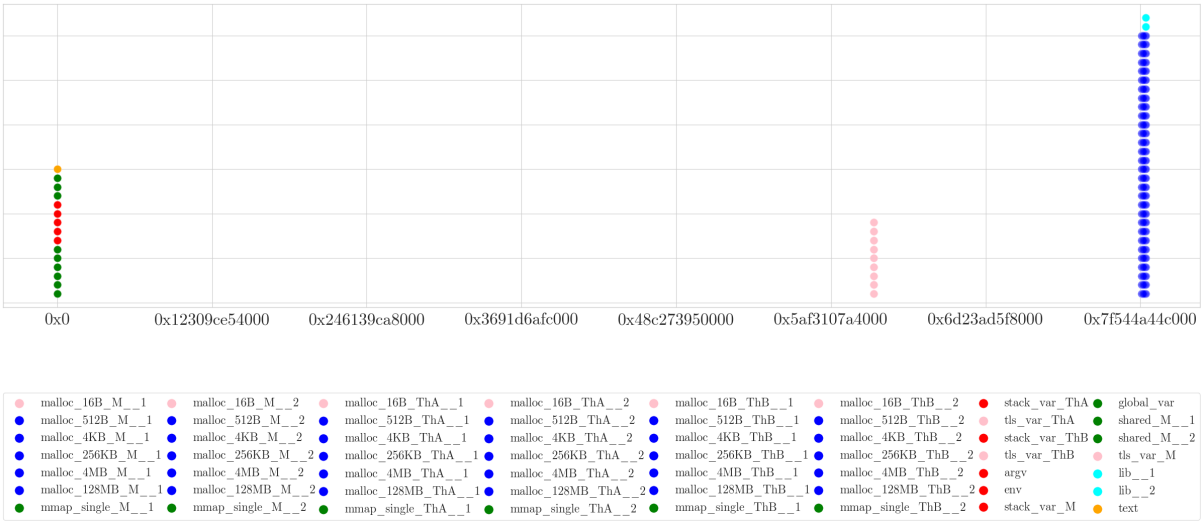

Figure 5.4: Memory Layout MacOS M1 Rosetta

	MIN	MAX	AVG		MIN	MAX	AVG
executable	0.00	13.58	4.53	16B	12.61	14.57	13.49
stack	14.67	15.12	14.82	512B	16.55	19.10	17.76
mmap()	13.58	13.59	13.59	4KB	16.51	18.50	17.45
main malloc()	12.00	16.55	14.30	256KB	12.00	14.26	13.14
thread malloc()	13.19	19.10	15.66	4MB	12.06	14.28	13.31
1st malloc()	12.00	18.17	15.00	128MB	16.04	16.40	16.17
2nd malloc()	12.13	19.10	15.44	shared	13.58	13.58	13.58
mmap() huge	NA	NA	NA	other	13.58	15.97	14.73

Table 5.13: Entropy Groups MacOS M1 Rosetta

Reboot Changes. The results presented in Table B.4 are comparable to the Native one discussed in Section 5.2.1, with 12.2 bit of absolute entropy in libraries and no change in other objects. Even the Probability Distributions of the newly randomized sections presented in Figure B.3 are comparable. Again, the randomization process performed during reboot is insufficient in entropy to provide any sort of real protection.

5.3. Windows

Sampling of Windows system was done inside an emulated environment running on a machine configurated as in Table 5.15 . Moreover, for Windows 11 sampling the configuration adopted is provided in Table 5.16.

path	ent	diff	path	ent	diff
text ← shared	0.00	-13.58	text ← mmap_single_Th	1.19	-12.39
text ← global_var	0.00	-13.58	text ← mmap_single_M	0.00	-13.58

Table 5.14: Positive Correlation executable path MacOS M1 Rosetta

Hypervisor	QEMU 7.1.0
CPU	i7-4790K
RAM	32GB RAM

Table 5.15: General Configuration Windows

5.3.1. Windows 11

Memory Layout Figure 5.5:
Looking at the memory layout we clearly identify 3 regions. The first two are located at low addresses, one hosts all `malloc()` objects and `VirtualAlloc()` objects, and the other hosts all stacks, both thread and main ones. At the other end of the memory, we can see global variables, libraries, and text area, which are the objects whose position is fixed inside the memory.

Probability Distribution. In Figure C.1 we can observe 3 different shapes for the distribution. We have stacks that are randomized uniformly, text, libraries and global variables that are fixed, and `malloc()` 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 what we call an Irwin–Hall distribution. This choice, even if provides a larger entropy compared to 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 5.17 we see that overall randomization entropy almost every time over 23 bit, with some objects reaching as high as 27 bit. As expected huge pages stop at less entropy compared to other sections, due to larger page offset. From Table C.1 were we provided the whole results, we confirm that global variables, libraries, and text are fixed in position and pose major risks.

Correlation Entropy. The correlation matrix shown in Figure C.2 tells what already partially emerged from the memory layout: High correlation inside the identified areas and a low correlation between areas. However, no relevant Positive Correlation Path emerged as the position of executable objects are already fixed.

Reboot Changes. The rebooting process randomizes also the fixed section. after per-

OS	Windows 11
Architecture	x86_64
Version	10.0.22621
First Boot Samples	3,800,000
Reboots	10,000
Reboot Samples	400
Total Reboot Samples	4,000,000

Table 5.16: Information Windows 11 Sampling

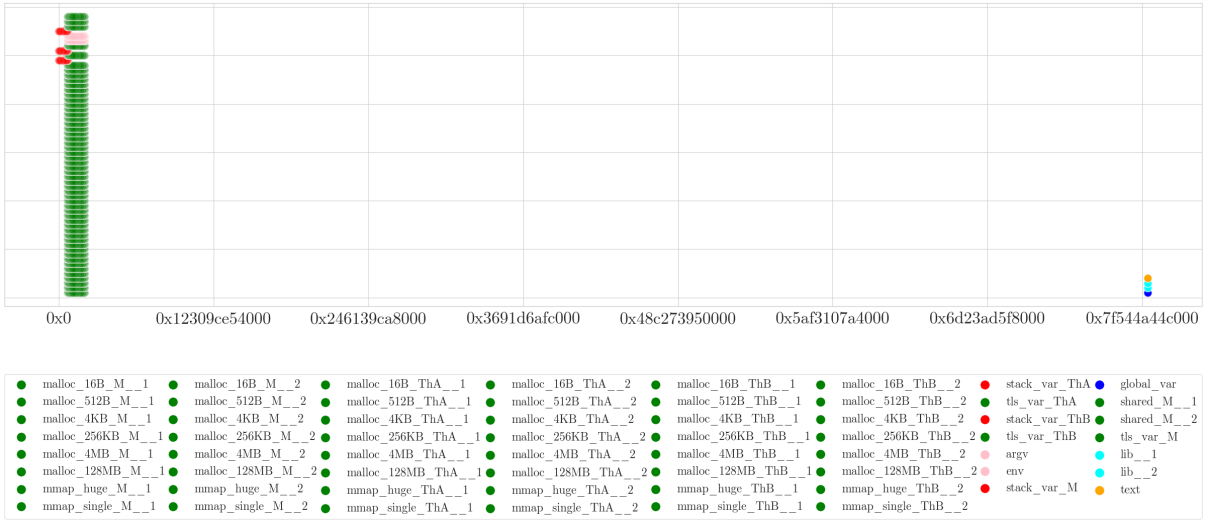

Figure 5.5: Memory Layout Windows 11

forming 10,000 reboots we can plot the corresponding Probability Distribution, in Figure C.3, and also calculate the difference in Absolute Entropy compared to the single boot scenario. The distribution appears to be uniform and the reboot seems to leave the entropy of the usually randomized sections untouched. Clearly from Table C.3, significant changes apply to global var, libraries, and text sections that now experience 13.2 bit of entropy. Still insufficient to protect against a brute-force attack.

5.4. Android

Android sampling was performed using Android Emulator with the configuration provided in Table 5.18. For the specific OS version considered, Android 13.0, the relevant configuration such as the number of reboots and number of samples are available in Table 5.19.

	MIN	MAX	AVG		MIN	MAX	AVG
executable	0.00	0.00	0.00	16B	27.41	27.41	27.41
stack	27.41	27.41	27.41	512B	25.91	27.41	27.16
mmap()	25.08	25.24	25.16	4KB	25.91	27.41	27.16
main malloc()	25.36	27.41	26.26	256KB	25.36	27.41	26.61
thread malloc()	22.39	27.41	26.01	4MB	23.84	27.41	24.80
1st malloc()	23.06	27.41	26.13	128MB	22.39	25.53	23.33
2nd malloc()	22.39	27.41	26.03	shared	25.34	25.35	25.35
mmap() huge	18.63	19.65	19.10	other	0.00	27.41	21.92

Table 5.17: Entropy Groups Windows 11

OS	Android 13.0
Architecture	arm64
Hypervisor	Android Emulator 17.0.6
Virtual Device	Google Pixel 6
Hardware	ARM M1

Table 5.18: General Configuration Android

5.4.1. Android 13

Memory Layout. As we expected after our research to identify the best allocation sizes, the allocated objects represented in Figure 5.6 are quite sparse inside the memory, with a trend towards lower addresses. On the right side, we can see the stack and global variable.

Probability Distribution. The graphs in Figure D.1 presents many spikes, suggesting little to no randomization in most of the sections, with only allocations of large sizes (128MB) done from the main having some sort of distribution. Also, stack variable, environment variables, and libraries are completely fixed.

Absolute Entropy. Our suspect of no-randomization in most of the sections is confirmed by the Absolute Entropy results, summarized in Table 5.20; clearly, only allocation greater than 4MB has been randomized while others are basically fixed in position. From the complete results available in Table D.1, we can see that global variables and text are randomized with 13 bit of entropy while libraries and main stack is fixed in position. Overall, the randomization is scarce as we expected.

Correlation Entropy. Being every executable object almost fixed, Correlation Entropy isn't as useful as in other OS, however we can identify a Positive Correlation Path from global variables to text area, visible in Table 5.21, that are completely correlated in position. The complete results are available in Figure D.2.

First Boot Samples	10,000
Reboots Sample	140
Reboots	3,150
Total Reboot Samples	441,00

Table 5.19: Information Android 13.0 Sampling

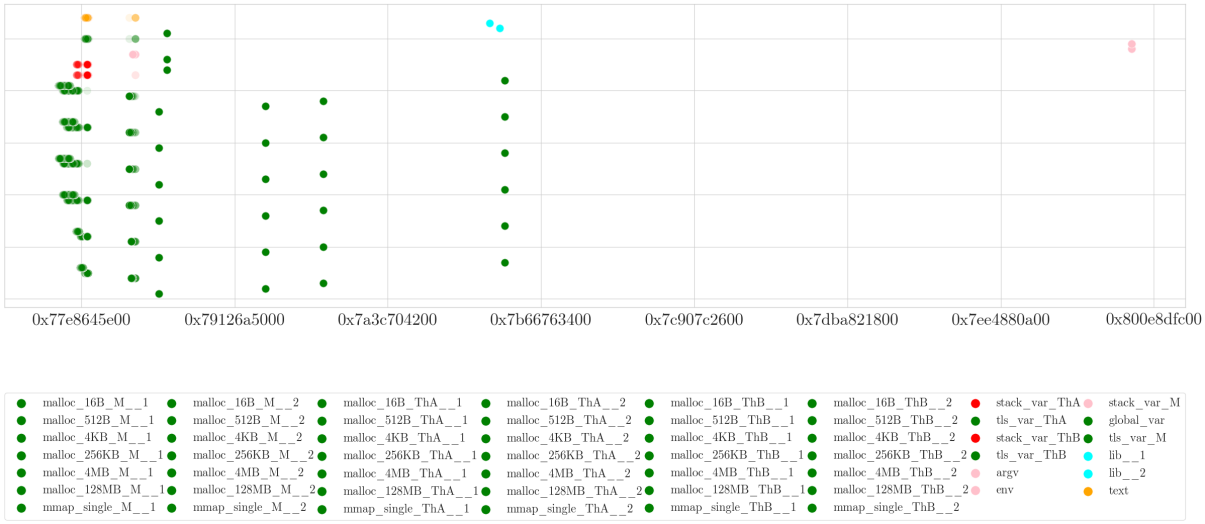

Figure 5.6: Memory Layout Android 13

Reboot Changes. In Table D.2 can see the resulting Absolute Entropy after reboot. Rebooting increases significantly the entropy of every object, at least to a detectable level. The objects that before were randomized to a detectable level are now at a significant level of entropy (>19.5 bit) while the other allocations range from 10 bit to 16 bit. Badly, libraries are randomized only with 9.8 bit of entropy, too few to contrast even the slowest attack.

	MIN	MAX	AVG		MIN	MAX	AVG
executable	0.00	13.10	4.37	16B	4.19	5.74	5.10
stack	0.00	7.80	5.20	512B	2.19	3.12	2.77
mmap()	0.80	1.79	1.44	4KB	0.43	1.86	1.35
main malloc()	0.36	14.01	4.85	256KB	0.04	1.40	0.94
thread malloc()	1.23	15.94	6.77	4MB	7.80	15.94	11.80
1st malloc()	0.36	15.34	5.94	128MB	14.01	15.34	14.86
2nd malloc()	0.04	15.94	6.34	shared	NA	NA	NA
mmap() huge	NA	NA	NA	other	0.00	13.10	4.86

Table 5.20: Entropy Groups Android 13.0

path	ent	diff	path	ent	diff
text←global_var	0.00	-13.10	text←malloc_128MB_M	9.55	-3.55

Table 5.21: Positive Correlation executable path Android 13.0

6 | Weakness and Attack POC

In this chapter, we will discuss some of the possibilities in terms of attack scenarios that are provided by the found vulnerability. after that, we will try to exploit them in a proof of concept to verify that our estimations were indeed true.

6.1. Weakness and Attacker Profile

Our analysis identified many lacking points in current ASLR implementations and we can categorize them and rank them based on which object they affect. Even if every unrandomized one is a vulnerability, some are more important than others. For instance, to build a successful ROP we have to correctly guess the position of gadgets inside memory, so randomizing this type of object is the priority. On the other hand, knowing the position of a malloc object might be useful to read its content or write information, but by themself cannot directly be exploited to execute code. However, they might suggest the position of executable objects due to Positive Correlation.

We need to define two attacker profile to better evaluate the severity of the problems found:

- **Local attacker without information disclosure**: The attacker has access to both the system and the vulnerable program.
- **Local attacker with information disclosure**: The attacker has access to the system and the vulnerable program and the program leaks the address of an arbitrary non-executable object.
- **Remote attacker without information disclosure**: The attacker has access only to the program and does not know the memory layout of other programs on the machine.
- **Remote attacker with information disclosure**: The attacker has access only to the program and the program leaks the address of an arbitrary non-executable object.

In all these situations we assume the attacker able to redirect the flow of the execution, so the stack smashing protection is defeated or disabled in the first place. According to these considerations, we can rank the weaknesses found in the analysis process from the most severe to the least severe:

1. Fixed position of executable objects
2. Low entropy of executable objects in different reboot
3. Low entropy of executable objects in a single reboot
4. Presence of Positive Correlation paths to executable objects
5. General low entropy

6.2. Attack Scenarios

Based on the highlighted weaknesses and the different attacker profiles we can define 3 main distinct attack scenarios.

6.2.1. Fixed Position

The most severe problem is the fixed position of executable objects: .text and libraries. Those objects can be used to build ROP and potentially execute malicious code; in this situation, a Local Attacker could launch another program, gather the information about address libraries of the current system configuration, and then use this address to build a ROP in just one try. In case another program cannot be launched and the current system configuration is unknown Local Attackers and Remote Attackers without leaks are in the same situation: brute-forcing the address position; at this point, the effort we need to put in is half the reboot entropy, in fact, if the reboot entropy of an executable object is 13.2 bit as in Windows 11, on average we will need $\frac{2^{13.2}}{2} = 4,700$ tries to correctly guess the address. In the presence of a leak, we could exploit the **Positive Correlation** paths; for instance, in MacOS M1 in the same boot, a leaked address of a `malloc()` object of sizes between 512B and 128MB reduces the entropy of libraries from 12 bit to 7 bit.

6.2.2. Random Position

If the position of executable objects is randomized at every program launch, like in Linux, we cannot use information gathered from other programs in a local attack, and the only possible approach is to directly brute force the object. In Linux the entropy of executable

objects is over 27 bit so an attack of those types is hard to achieve in a reasonable time. However, since the Linux version 5.18 the entropy of libraries of sizes greater than 2MB was drastically reduced. This is a major concern since now is under the threshold of 20 bit. If we have a leak of an address both a local and remote attacker could exploit one of the many Positive Correlation path present in Linux. A leak of a `sbrk()` object will reduce the entropy of text from 27 bit to just 13 bit and a leak in other `malloc()` will reduce the entropy of libraries from 27 bit to 9 bit in the best case, to 1 in the worst.

6.2.3. Distributed Attack

Another scenario is the distributed attack. Imagine a vulnerability in a wide diffused tool (like the one found in log4j). If we can target multiple devices the reboot entropy will tell us how many hits we will collect after one try. In fact, if an object has an entropy S the probability of a device being in a specific state A, the one we write in the exploit, is P(A) = 1/S, so if N devices are infected we should expect N*P(A)=N/S device to be in the chosen configuration. For instance, a vulnerability of this type affecting 10,000 MacOS systems that have a reboot entropy of libraries of 12.3 bit, means that on one try we should hit around 2 systems.

6.3. Positive Correlation Attack POC

To validate our findings about the Positive Correlation path in Linux Kernels we set up a dummy program with an explicit buffer overflow vulnerability and a leak of a malloc() object to reduce the entropy of text to around 13 bit. The vulnerable program is presented in Algorithm 6.1 and was compiled using `gcc -fno-stack-protector -o vuln vuln.c` to disable the stack protection and enable the buffer overflow. The attack was performed using a Python script using the library `pwntools` that launched the vulnerable program, gathered the leak, added a fixed offset found doing dynamic analysis on the program, launched the exploit targeting the `flag()` function inside the text object. After that, the script observed the output. If it was "FLAG", then the exploit was successful, otherwise, we missed the target. We repeated the simulation on both Linux 5.17 and Linux 6.4.9 for a total of 2Mln tries. In the end, we had 261 successes, averaging one success every 7650 tries. This translates into an attack complexity of $2^{12.9bit}$ compared to an expected 12.99 bit of entropy. This is a difference of 0.753% in entropy from the real value, so our estimation was revealed to be solid.

Algorithm 6.1 Vulnerable Program

```
1: function FLAG
2:
3:     OUTPUT("FLAG")
4:
5: end function
6:
7: function VULN_FUNCTION
8:
9:     NAME[16]                                   ▷ vulnerable buffer
10:    HEAP_TARGET ← MALLOC(4KB)
11:
12:    OUTPUT("the target is " + HEAP_TARGET)
13:    GETS(NAME)                                 ▷ buffer overflow
14:
15: end function
16:
```

7 | Conclusions and future developments

In this thesis, the effectiveness of Address Space Layout Randomization was evaluated across major desktop and mobile platforms through statistical analysis of memory object position. The choice of a low-bias estimator like NSB permitted us to reduce the number of collected samples, allowing the analysis of even slow processes like device reboot.
We highlight major problems in the performance of ASLR systems. In some cases long-standing, like the lack of entropy of libraries and executable objects in Windows, MacOS, and Android. Other problems were newly introduced like the reduction found in recent Linux distributions. Overall, Linux distributions provide good randomization, while Windows, MacOS, and Android fail to adequately randomize key memory areas like executable code and libraries. Positive correlations between objects on Linux reduce entropy to dangerously low levels that our proof-of-concept exploits validated. The 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. Even if some solutions like ASLR-NG [14] were proposed in the years, claiming to improve drastically the effectiveness of ASLR on Linux systems, their current implementation is unavailable. Moreover, this research suggests that the evolution of operating systems often is 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 [25]. Another aspect underlined 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 the expectation, often lower.
According to these findings, future work includes the realization of a software profiler that analyzes the memory mappings of real software and their allocated object. Also, the extension of this research to other Linux security-hardened distributions could provide comparative insight into the choice of enterprise operating systems.

Bibliography

[1] J. Acharya, A. Orlitsky, A. T. Suresh, and H. Tyagi. Estimating renyi entropy of discrete distributions. *IEEE Transactions on Information Theory*, 63(1):38–56, 2017. doi: 10.1109/TIT.2016.2620435.

[2] D. H. Aristizabal, D. M. Rodriguez, and R. Y. Guevara. Measuring aslr implementations on modern operating systems. In *2013 47th International Carnahan Conference on Security Technology (ICCST)*, pages 1–6, 2013. doi: 10.1109/CCST.2013.6922073.

[3] Y. Ding, Z. Peng, Y. Zhou, and C. Zhang. Android low entropy demystified. In *2014 IEEE International Conference on Communications (ICC)*, pages 659–664, 2014. doi: 10.1109/ICC.2014.6883394.

[4] R. V. Díaz, M. Rivera-Dourado, R. Pérez-Jove, P. V. Avendaño, and J. M. Vázquez-Naya. Aslr analyzer, 2019. URL `https://github.com/raquelvqz/aslr/`.

[5] R. V. Díaz, M. Rivera-Dourado, R. Pérez-Jove, P. V. Avendaño, and J. M. Vázquez-Naya. Address space layout randomization comparative analysis on windows 10 and ubuntu 18.04 lts †. *Engineering Proceedings*, 7, 2021. ISSN 26734591. doi: 10.3390/engproc2021007026.

[6] W. Herlands, T. Hobson, and P. J. Donovan. Effective entropy: Security-Centric metric for memory randomization techniques. In *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, San Diego, CA, aug 2014. USENIX Association. URL `https://www.usenix.org/conference/cset14/workshop-program/presentation/herlands`.

[7] D. G. Hernández and I. Samengo. Estimating the mutual information between two discrete, asymmetric variables with limited samples. *Entropy*, 21, 6 2019. ISSN 10994300. doi: 10.3390/e21060623.

[8] D. Kaplan, S. Kedmi, R. Hay, and A. Dayan. Attacking the linux prng on android: Weaknesses in seeding of entropic pools and low boot-time entropy. In *Proceedings of the 8th USENIX Conference on Offensive Technologies*, WOOT'14, page 14, USA, 2014. USENIX Association.

[9] M. Kerrisk. pthread_create(3) - Linux Manual, Jan 2023. URL https://man7.org/linux/man-pages/man3/pthread_create.3.html.

[10] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened aslr on android. pages 424–439. Institute of Electrical and Electronics Engineers Inc., 11 2014. ISBN 9781479946860. doi: 10.1109/SP.2014.34.

[11] S. Liebergeld and M. Lange. *Android Security, Pitfalls and Lessons Learned*, volume 264, pages 409–417. 09 2013. ISBN 978-3-319-01603-0. doi: 10.1007/978-3-319-01604-7_40.

[12] H. Marco Gisbert and I. Ripoli. On the effectiveness of full-aslr on 64-bit linux. Nov. 2014. URL https://deepsec.net/archive/2014.deepsec.net/index.html. In-depth Security Conference 2014 (DeepSec) ; Conference date: 18-11-2014 Through 21-11-2014.

[13] H. Marco Gisbert and I. Ripoll. Exploiting linux and pax aslr's weaknesses on 32- bit and 64-bit systems. Mar. 2016. URL https://www.blackhat.com/asia-16/briefings.html. Black Hat Asia 2016 ; Conference date: 29-03-2016 Through 01-04-2016.

[14] H. Marco-Gisbert and I. Ripoll Ripoll. Address space layout randomization next generation. *Applied Sciences*, 9(14), 2019. ISSN 2076-3417. doi: 10.3390/app9142928. URL https://www.mdpi.com/2076-3417/9/14/2928.

[15] S. Marsili and C. Cattuto. Nsb entropy estimator, python implementation, 2021. URL https://github.com/simomarsili/ndd/tree/master.

[16] I. Nemenman. NSB Entropy Estimator, 2012. URL https://nemenmanlab.org/~ilya/index.php/Entropy_Estimation.

[17] I. Nemenman, F. Shafee, and W. Bialek. Entropy and inference, revisited. In *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2001. URL https://proceedings.neurips.cc/paper_files/paper/2001/file/d46e1fcf4c07ce4a69ee07e4134bcef1-Paper.pdf.

[18] Security Scorecard. Cve security vulnerability database. security vulnerabilities, exploits, Aug 2023. URL https://www.cvedetails.com/vulnerabilities-by-types.php.

[19] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[20] Statcounter Global Stats. Desktop Operating System Market Share Worldwide, Aug 2023. URL https://gs.statcounter.com/os-market-share/desktop/worldwide.

[21] Statcounter Global Stats. Mobile Operating System Market Share Worldwide, Aug 2023. URL https://gs.statcounter.com/os-market-share/mobile/worldwide.

[22] The kernel development community. Libc: Malloc tunable parameters, Jan 2023. URL https://www.gnu.org/software/libc/manual/2.37/html_mono/libc.html#Malloc-Tunable-Parameters.

[23] The kernel development community. Memory management apis documentation, Jan 2023. URL https://www.kernel.org/doc/html/v6.0/core-api/mm-api.html.

[24] L. Torvalds. Linux 5.15 x86 repository, Jan 2021. URL https://github.com/torvalds/linux/tree/v5.15/arch/x86.

[25] L. Torvalds. Memory management in linux 5.15, Aug 2021. URL https://github.com/torvalds/linux/blob/v5.15/Documentation/x86/x86_64/mm.rst.

[26] W3Techs. Linux market share, Jan 2023. URL https://truelist.co/blog/linux-statistics/.

[27] W3Techs. Usage statistics of operating systems for websites, Sep 2023. URL https://w3techs.com/technologies/overview/operating_system.

[28] O. Whitehouse. An analysis of address space layout randomization on windows vista. 2007. URL https://api.semanticscholar.org/CorpusID:62370377.

[29] A. abrocki. Scraps of notes on remote stack overflow exploitation, Nov 2010. URL http://phrack.org/issues/67/13.html#article.

A | Linux Results

Object	Ent	Object	Ent
malloc_16B_M__1	27.409	mmap_single_ThA__2	27.409
malloc_512B_M__1	27.409	malloc_16B_ThB__1	14.744
malloc_4KB_M__1	27.409	malloc_512B_ThB__1	14.744
malloc_256KB_M__1	27.408	malloc_4KB_ThB__1	14.744
malloc_4MB_M__1	27.408	malloc_256KB_ThB__1	24.773
malloc_128MB_M__1	27.408	malloc_4MB_ThB__1	19.912
mmap_huge_M__1	19.024	malloc_128MB_ThB__1	15.660
mmap_single_M__1	27.408	mmap_huge_ThB__1	18.869
malloc_16B_M__2	27.409	mmap_single_ThB__1	27.408
malloc_512B_M__2	27.409	malloc_16B_ThB__2	14.744
malloc_4KB_M__2	27.409	malloc_512B_ThB__2	14.744
malloc_256KB_M__2	25.911	malloc_4KB_ThB__2	14.744
malloc_4MB_M__2	19.588	malloc_256KB_ThB__2	24.154
malloc_128MB_M__2	19.588	malloc_4MB_ThB__2	18.977
mmap_huge_M__2	19.024	malloc_128MB_ThB__2	16.227
mmap_single_M__2	27.408	mmap_huge_ThB__2	18.787
malloc_16B_ThA__1	14.926	mmap_single_ThB__2	27.408
malloc_512B_ThA__1	14.926	stack_var_ThA	19.024
malloc_4KB_ThA__1	14.926	tls_var_ThA	19.024
malloc_256KB_ThA__1	24.247	stack_var_ThB	19.026
malloc_4MB_ThA__1	19.752	tls_var_ThB	19.026
malloc_128MB_ThA__1	16.236	argv	27.395
mmap_huge_ThA__1	18.814	env	22.472
mmap_single_ThA__1	27.409	stack_var_M	27.395
malloc_16B_ThA__2	14.926	global_var	27.409
malloc_512B_ThA__2	14.926	shared_M__1	27.408
malloc_4KB_ThA__2	14.926	shared_M__2	27.408
malloc_256KB_ThA__2	23.722	tls_var_M	27.408
malloc_4MB_ThA__2	18.929	lib__1	27.408
malloc_128MB_ThA__2	16.840	lib__2	27.408
mmap_huge_ThA__2	18.711	text	27.409

Table A.1: Absolute Entropy Linux 5.17.15

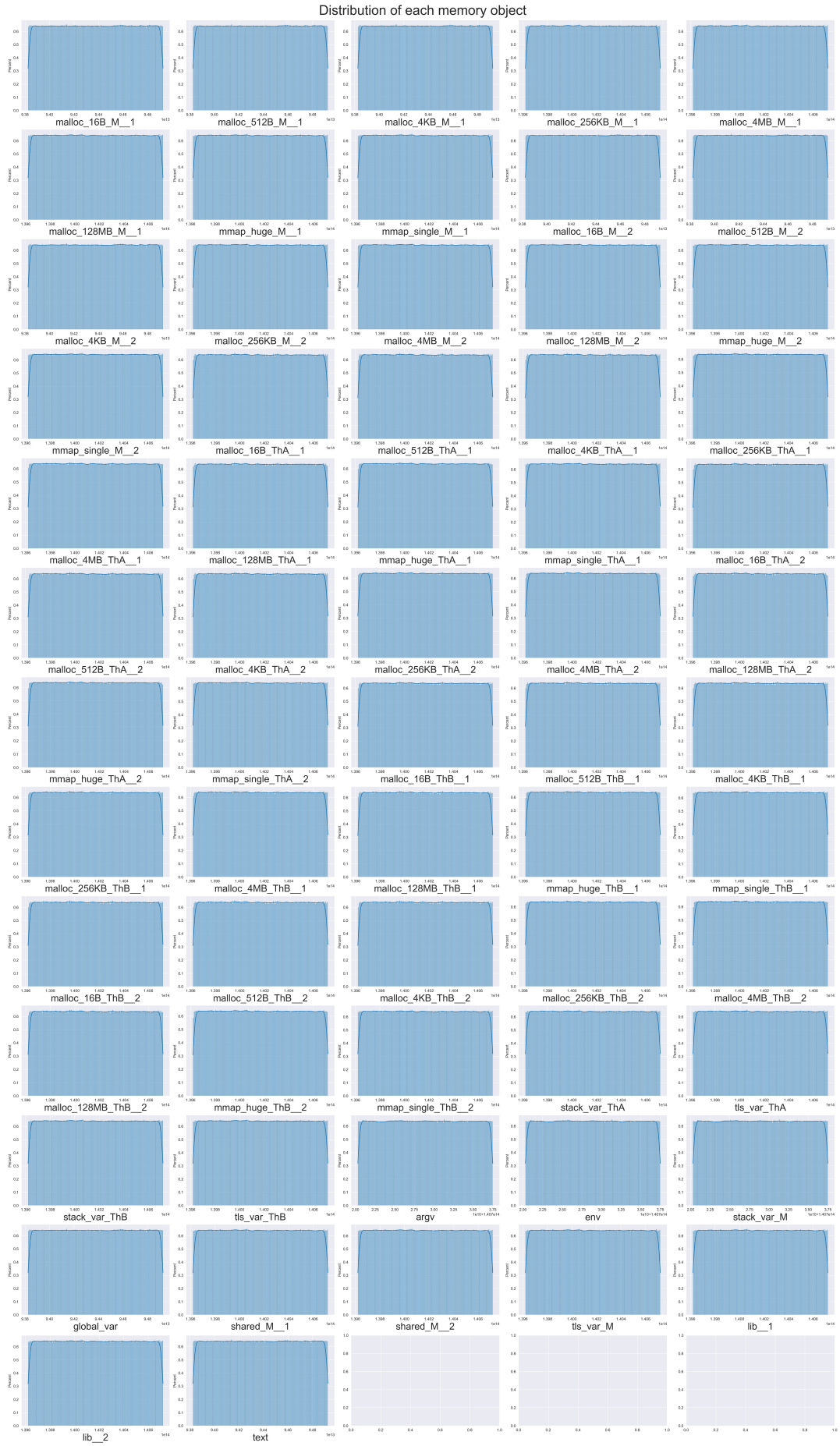

Figure A.1: Probability Distribution Linux 5.17.19

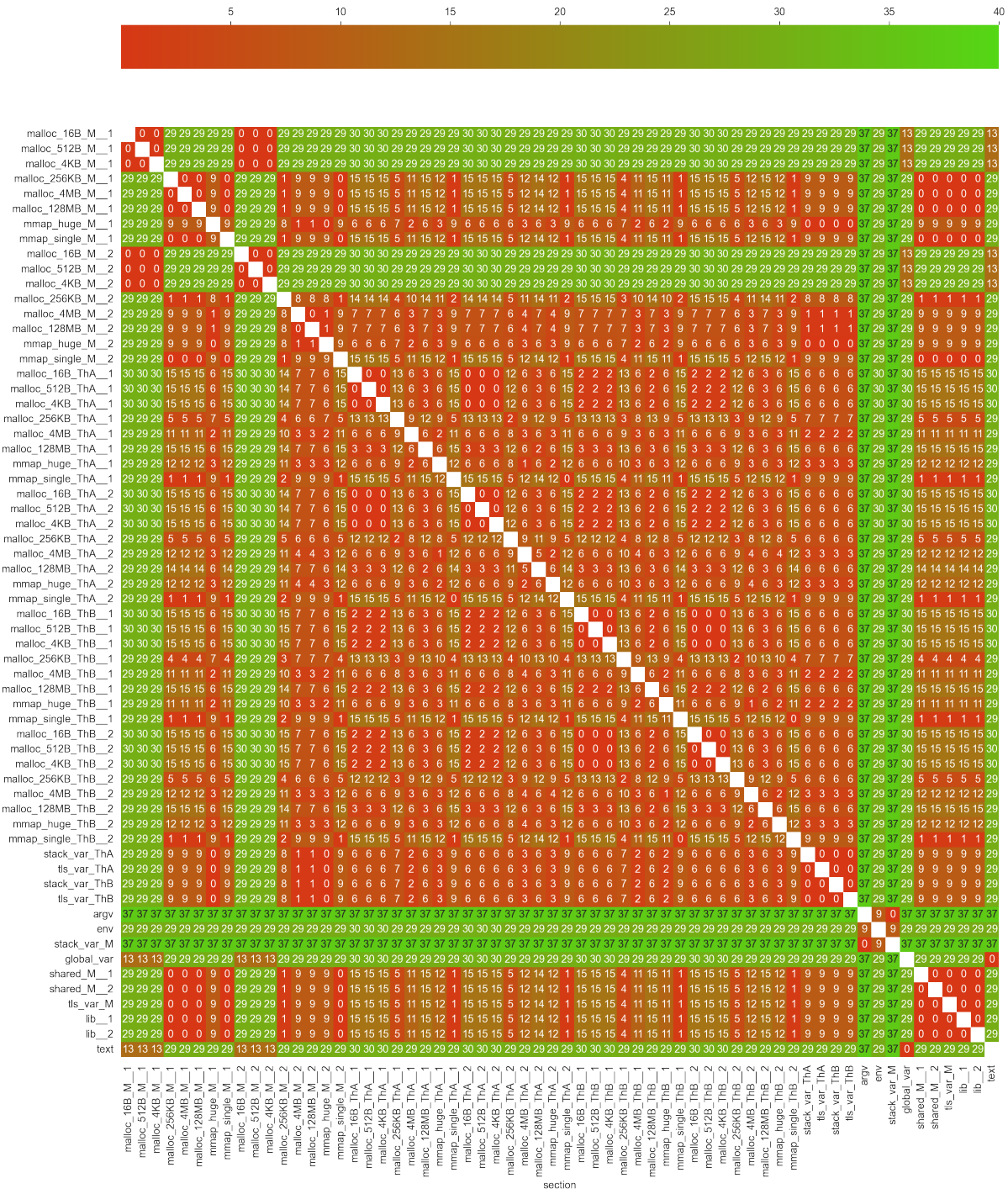

Figure A.2: Correlation Entropy Linux 5.17.19

Object	Ent	Object	Ent
malloc_16B_M__1	27.409	mmap_single_ThA__2	27.409
malloc_512B_M__1	27.409	malloc_16B_ThB__1	14.931
malloc_4KB_M__1	27.409	malloc_512B_ThB__1	14.931
malloc_256KB_M__1	25.832	malloc_4KB_ThB__1	14.931
malloc_4MB_M__1	19.611	malloc_256KB_ThB__1	24.006
malloc_128MB_M__1	19.611	malloc_4MB_ThB__1	20.197
mmap_huge_M__1	19.023	malloc_128MB_ThB__1	16.134
mmap_single_M__1	27.409	mmap_huge_ThB__1	18.883
malloc_16B_M__2	27.409	mmap_single_ThB__1	27.409
malloc_512B_M__2	27.409	malloc_16B_ThB__2	14.931
malloc_4KB_M__2	27.409	malloc_512B_ThB__2	14.931
malloc_256KB_M__2	24.902	malloc_4KB_ThB__2	14.931
malloc_4MB_M__2	19.023	malloc_256KB_ThB__2	23.522
malloc_128MB_M__2	19.023	malloc_4MB_ThB__2	19.125
mmap_huge_M__2	19.023	malloc_128MB_ThB__2	16.636
mmap_single_M__2	27.409	mmap_huge_ThB__2	18.808
malloc_16B_ThA__1	14.978	mmap_single_ThB__2	27.409
malloc_512B_ThA__1	14.978	stack_var_ThA	19.023
malloc_4KB_ThA__1	14.978	tls_var_ThA	19.023
malloc_256KB_ThA__1	23.998	stack_var_ThB	19.029
malloc_4MB_ThA__1	20.007	tls_var_ThB	19.029
malloc_128MB_ThA__1	16.071	argv	27.395
mmap_huge_ThA__1	18.875	env	22.472
mmap_single_ThA__1	27.409	stack_var_M	27.395
malloc_16B_ThA__2	14.978	global_var	27.409
malloc_512B_ThA__2	14.978	shared_M__1	27.409
malloc_4KB_ThA__2	14.978	shared_M__2	27.409
malloc_256KB_ThA__2	23.501	tls_var_M	27.409
malloc_4MB_ThA__2	19.117	lib__1	19.023
malloc_128MB_ThA__2	16.786	lib__2	27.409
mmap_huge_ThA__2	18.804	text	27.409

Table A.2: Absolute Entropy Linux 6.4.9

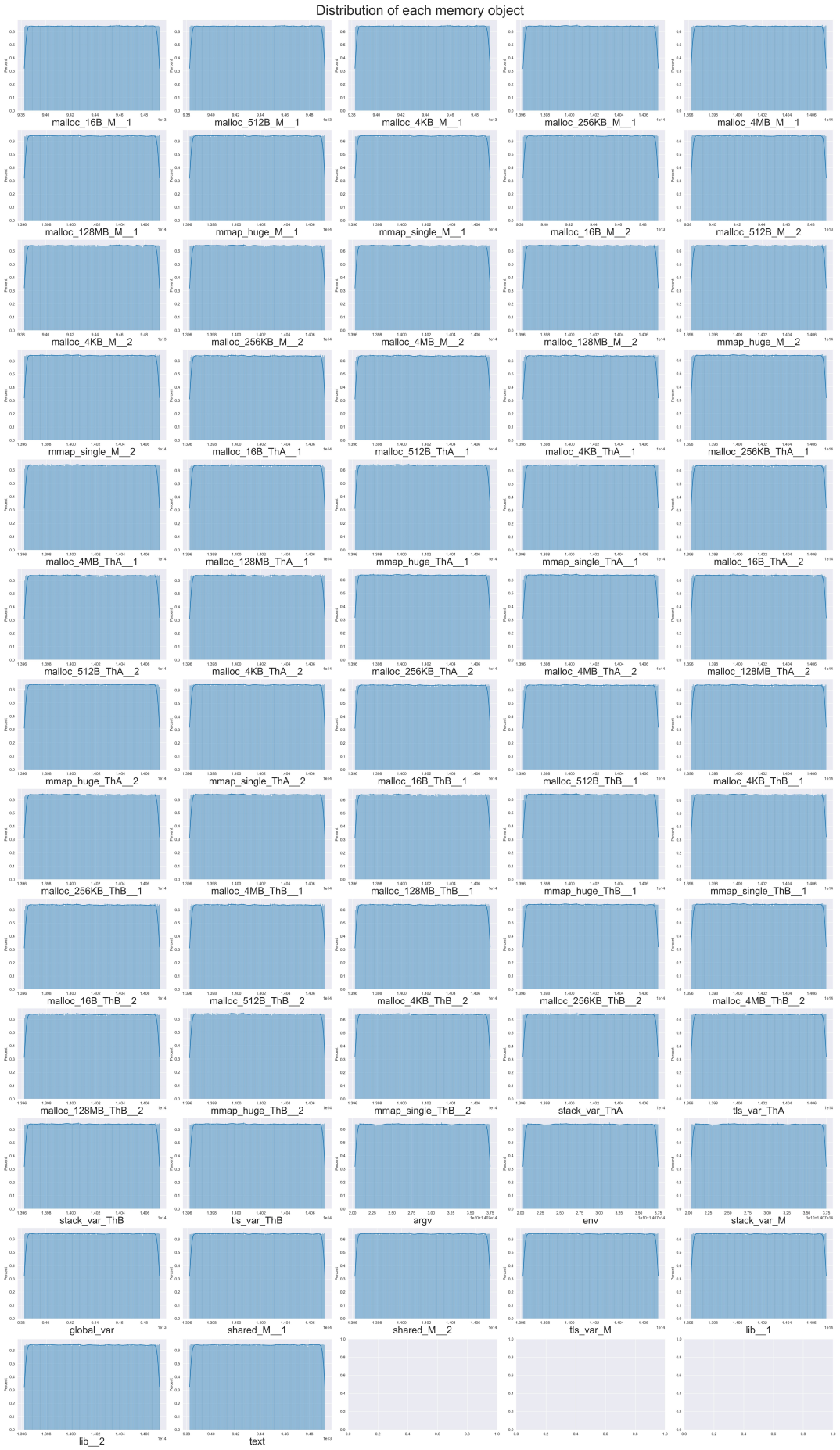

Figure A.3: Probability Distribution Linux 6.4.9

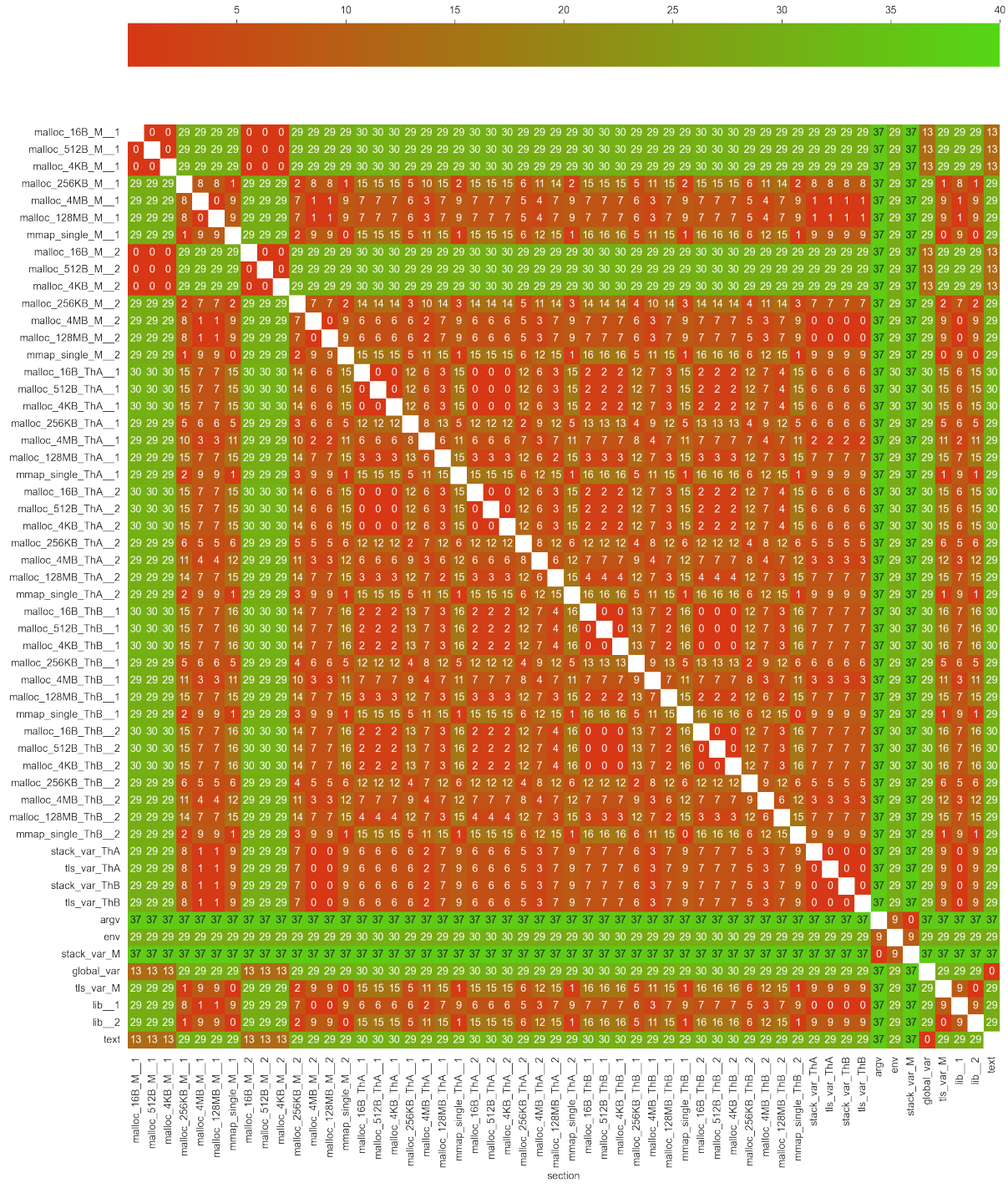

Figure A.4: Correlation Entropy Linux 6.4.9

Object	Ent	Object	Ent
malloc_16B_M__1	0.000	mmap_single_ThA__2	0.000
malloc_512B_M__1	0.000	malloc_16B_ThB__1	0.187
malloc_4KB_M__1	0.000	malloc_512B_ThB__1	0.187
malloc_256KB_M__1	-1.576	malloc_4KB_ThB__1	0.187
malloc_4MB_M__1	-7.797	malloc_256KB_ThB__1	-0.767
malloc_128MB_M__1	-7.797	malloc_4MB_ThB__1	0.286
mmap_huge_M__1	-0.000	malloc_128MB_ThB__1	0.474
mmap_single_M__1	0.001	mmap_huge_ThB__1	0.013
malloc_16B_M__2	0.000	mmap_single_ThB__1	0.001
malloc_512B_M__2	0.000	malloc_16B_ThB__2	0.187
malloc_4KB_M__2	0.000	malloc_512B_ThB__2	0.187
malloc_256KB_M__2	-1.010	malloc_4KB_ThB__2	0.187
malloc_4MB_M__2	-0.565	malloc_256KB_ThB__2	-0.632
malloc_128MB_M__2	-0.565	malloc_4MB_ThB__2	0.147
mmap_huge_M__2	-0.000	malloc_128MB_ThB__2	0.409
mmap_single_M__2	0.001	mmap_huge_ThB__2	0.021
malloc_16B_ThA__1	0.052	mmap_single_ThB__2	0.001
malloc_512B_ThA__1	0.052	stack_var_ThA	-0.000
malloc_4KB_ThA__1	0.052	tls_var_ThA	-0.000
malloc_256KB_ThA__1	-0.249	stack_var_ThB	0.003
malloc_4MB_ThA__1	0.254	tls_var_ThB	0.003
malloc_128MB_ThA__1	-0.165	argv	0.000
mmap_huge_ThA__1	0.061	env	-0.000
mmap_single_ThA__1	-0.000	stack_var_M	0.000
malloc_16B_ThA__2	0.052	global_var	-0.000
malloc_512B_ThA__2	0.052	shared_M__1	0.001
malloc_4KB_ThA__2	0.052	shared_M__2	0.001
malloc_256KB_ThA__2	-0.221	tls_var_M	0.001
malloc_4MB_ThA__2	0.188	lib__1	-8.385
malloc_128MB_ThA__2	-0.054	lib__2	0.001
mmap_huge_ThA__2	0.093	text	-0.000

Table A.3: Absolute Entropy Change Linux 5.17.15 to Linux 6.4.9

B | MacOS Results

Object	Ent	Object	Ent
malloc_16B_M__1	12.583	malloc_16B_ThB__1	14.053
malloc_512B_M__1	7.551	malloc_512B_ThB__1	9.394
malloc_4KB_M__1	7.465	malloc_4KB_ThB__1	8.710
malloc_256KB_M__1	3.189	malloc_256KB_ThB__1	4.755
malloc_4MB_M__1	3.231	malloc_4MB_ThB__1	5.159
malloc_128MB_M__1	7.106	malloc_128MB_ThB__1	7.485
mmap_single_M__1	11.587	mmap_single_ThB__1	12.455
malloc_16B_M__2	12.639	malloc_16B_ThB__2	14.554
malloc_512B_M__2	7.619	malloc_512B_ThB__2	9.939
malloc_4KB_M__2	7.533	malloc_4KB_ThB__2	9.180
malloc_256KB_M__2	3.274	malloc_256KB_ThB__2	5.349
malloc_4MB_M__2	3.278	malloc_4MB_ThB__2	5.360
malloc_128MB_M__2	7.857	malloc_128MB_ThB__2	7.162
mmap_single_M__2	12.046	mmap_single_ThB__2	12.582
malloc_16B_ThA__1	13.728	stack_var_ThA	11.583
malloc_512B_ThA__1	9.237	tls_var_ThA	13.160
malloc_4KB_ThA__1	8.484	stack_var_ThB	11.583
malloc_256KB_ThA__1	4.550	tls_var_ThB	13.521
malloc_4MB_ThA__1	5.038	argv	12.028
malloc_128MB_ThA__1	7.240	env	12.903
mmap_single_ThA__1	12.478	stack_var_M	11.820
malloc_16B_ThA__2	14.520	global_var	11.583
malloc_512B_ThA__2	9.984	shared_M__1	11.583
malloc_4KB_ThA__2	9.137	shared_M__2	11.583
malloc_256KB_ThA__2	5.282	tls_var_M	12.549
malloc_4MB_ThA__2	5.295	lib__1	0.000
malloc_128MB_ThA__2	6.650	lib__2	0.000
mmap_single_ThA__2	12.611	text	11.583

Table B.1: Absolute Entropy MacOS M1 Native

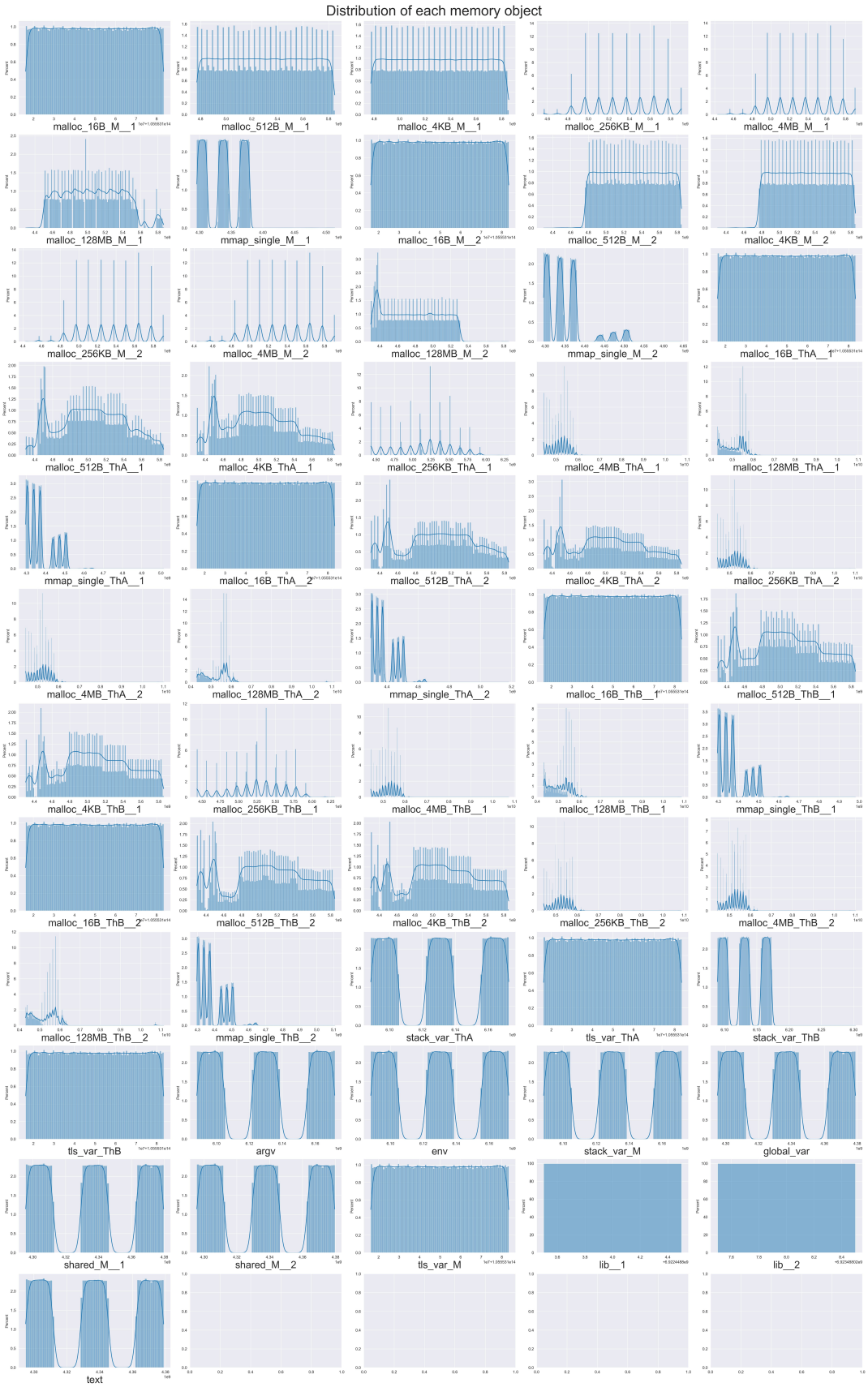

Figure B.1: Probability Distribution MacOS M1 Native

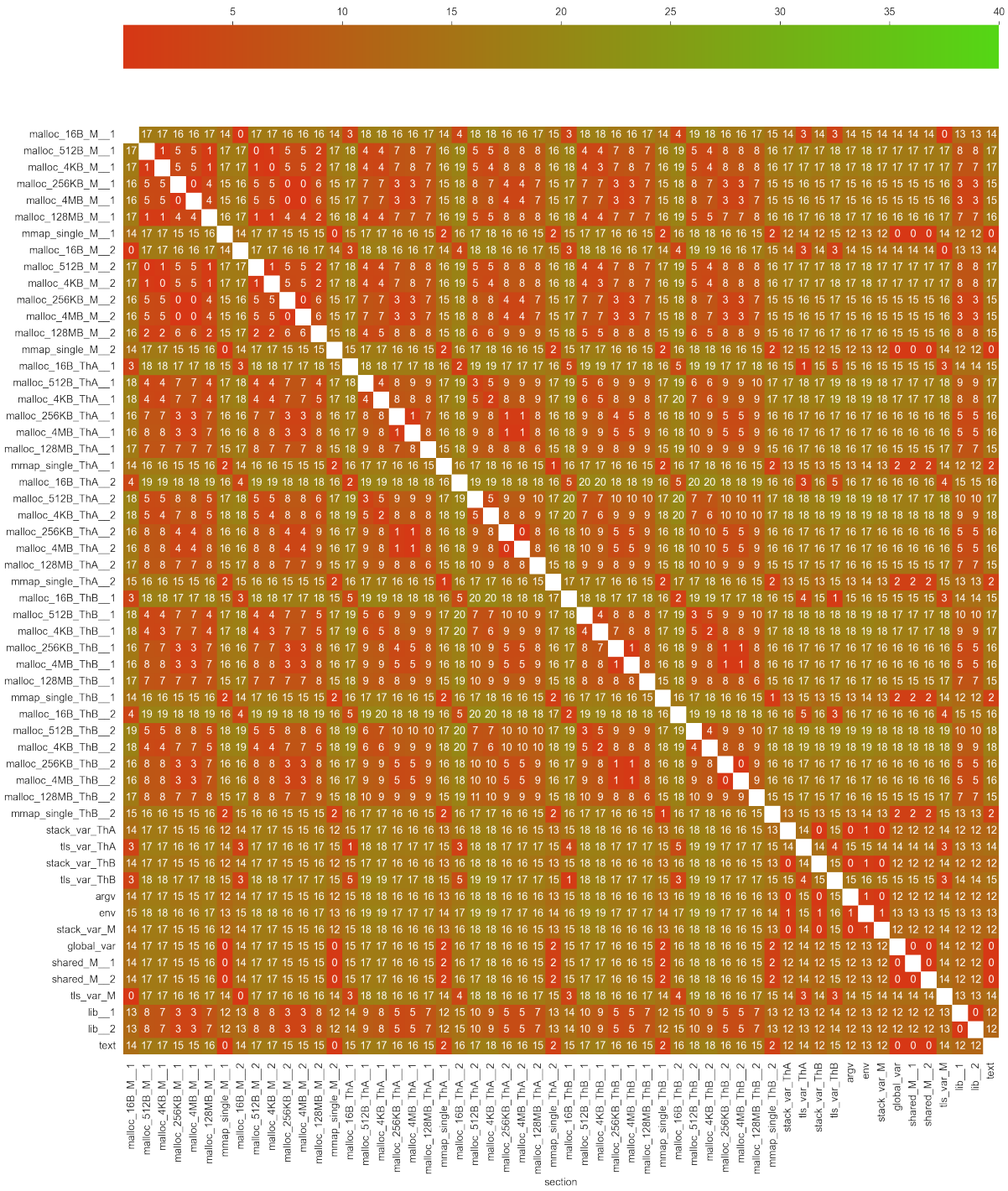

Figure B.2: Correlation Entropy MacOS M1 Native

Object	Ent	Object	Ent
malloc_16B_M__1	0.231	malloc_16B_ThB__1	0.101
malloc_512B_M__1	0.207	malloc_512B_ThB__1	0.319
malloc_4KB_M__1	0.144	malloc_4KB_ThB__1	0.138
malloc_256KB_M__1	0.004	malloc_256KB_ThB__1	0.129
malloc_4MB_M__1	0.022	malloc_4MB_ThB__1	0.093
malloc_128MB_M__1	0.017	malloc_128MB_ThB__1	-0.242
mmap_single_M__1	0.003	mmap_single_ThB__1	0.081
malloc_16B_M__2	0.266	malloc_16B_ThB__2	0.127
malloc_512B_M__2	0.250	malloc_512B_ThB__2	0.301
malloc_4KB_M__2	0.187	malloc_4KB_ThB__2	0.121
malloc_256KB_M__2	0.050	malloc_256KB_ThB__2	0.052
malloc_4MB_M__2	0.050	malloc_4MB_ThB__2	0.047
malloc_128MB_M__2	0.018	malloc_128MB_ThB__2	-0.513
mmap_single_M__2	-0.000	mmap_single_ThB__2	0.055
malloc_16B_ThA__1	0.071	stack_var_ThA	0.000
malloc_512B_ThA__1	0.017	tls_var_ThA	0.359
malloc_4KB_ThA__1	0.249	stack_var_ThB	0.000
malloc_256KB_ThA__1	0.053	tls_var_ThB	0.015
malloc_4MB_ThA__1	-0.123	argv	-0.376
malloc_128MB_ThA__1	0.228	env	-0.001
mmap_single_ThA__1	-0.081	stack_var_M	-0.208
malloc_16B_ThA__2	-0.123	global_var	0.000
malloc_512B_ThA__2	-0.169	shared_M__1	0.000
malloc_4KB_ThA__2	0.029	shared_M__2	0.000
malloc_256KB_ThA__2	-0.219	tls_var_M	0.217
malloc_4MB_ThA__2	-0.226	lib__1	12.281
malloc_128MB_ThA__2	0.311	lib__2	12.281
mmap_single_ThA__2	-0.049	text	0.000

Table B.2: Absolute Entropy Change MacOS M1 reboot

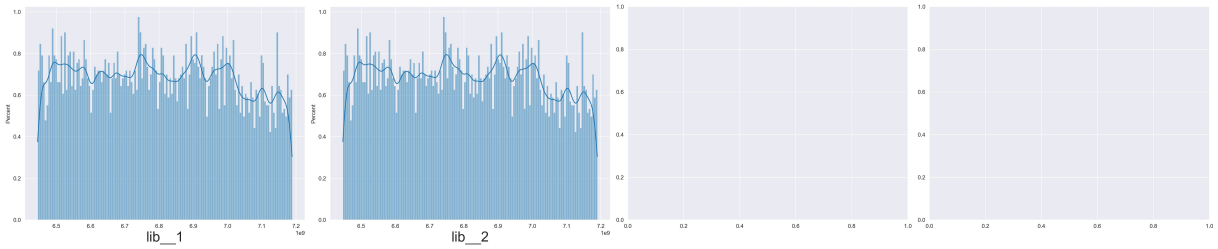

Figure B.3: Probability Distribution MacOS reboot new sections

Object	Ent	Object	Ent
malloc_16B_M__1	12.609	malloc_16B_ThB__1	13.516
malloc_512B_M__1	16.548	malloc_512B_ThB__1	18.169
malloc_4KB_M__1	16.506	malloc_4KB_ThB__1	17.632
malloc_256KB_M__1	12.000	malloc_256KB_ThB__1	13.398
malloc_4MB_M__1	12.059	malloc_4MB_ThB__1	13.942
malloc_128MB_M__1	16.070	malloc_128MB_ThB__1	16.095
mmap_single_M__1	13.584	mmap_single_ThB__1	13.586
malloc_16B_M__2	12.693	malloc_16B_ThB__2	14.575
malloc_512B_M__2	16.615	malloc_512B_ThB__2	19.102
malloc_4KB_M__2	16.582	malloc_4KB_ThB__2	18.501
malloc_256KB_M__2	12.129	malloc_256KB_ThB__2	14.257
malloc_4MB_M__2	12.133	malloc_4MB_ThB__2	14.278
malloc_128MB_M__2	16.041	malloc_128MB_ThB__2	16.404
mmap_single_M__2	13.584	mmap_single_ThB__2	13.585
malloc_16B_ThA__1	13.374	stack_var_ThA	14.674
malloc_512B_ThA__1	17.710	tls_var_ThA	14.336
malloc_4KB_ThA__1	17.414	stack_var_ThB	14.674
malloc_256KB_ThA__1	13.193	tls_var_ThB	14.722
malloc_4MB_ThA__1	13.600	argv	15.121
malloc_128MB_ThA__1	16.094	env	15.972
mmap_single_ThA__1	13.586	stack_var_M	15.121
malloc_16B_ThA__2	14.190	global_var	13.584
malloc_512B_ThA__2	18.438	shared_M__1	13.584
malloc_4KB_ThA__2	18.037	shared_M__2	13.584
malloc_256KB_ThA__2	13.844	tls_var_M	14.670
malloc_4MB_ThA__2	13.858	lib__1	0.000
malloc_128MB_ThA__2	16.329	lib__2	0.000
mmap_single_ThA__2	13.586	text	13.584

Table B.3: Absolute Entropy MacOS M1 Rosetta

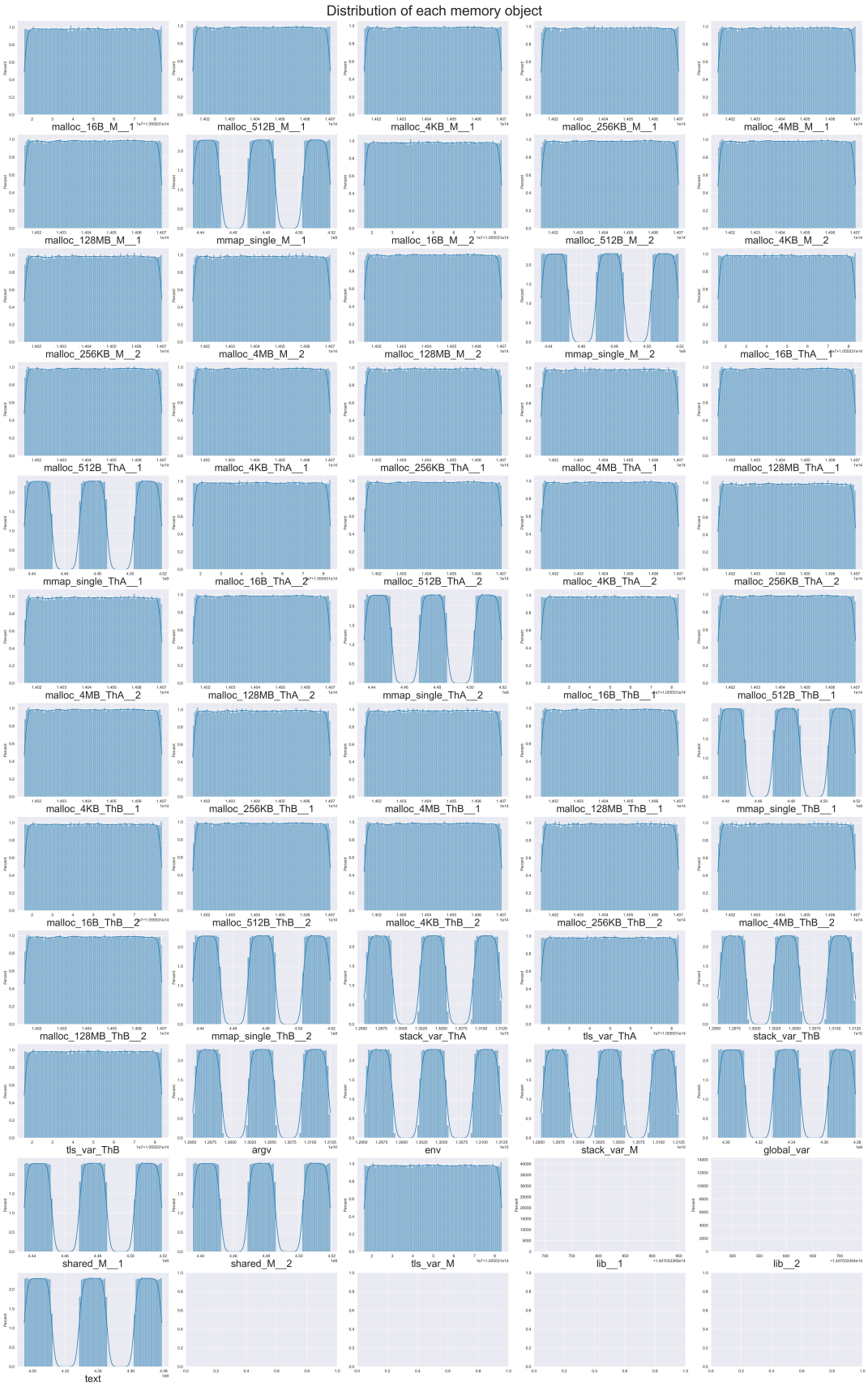

Figure B.4: Probability Distribution MacOS M1 Rosetta

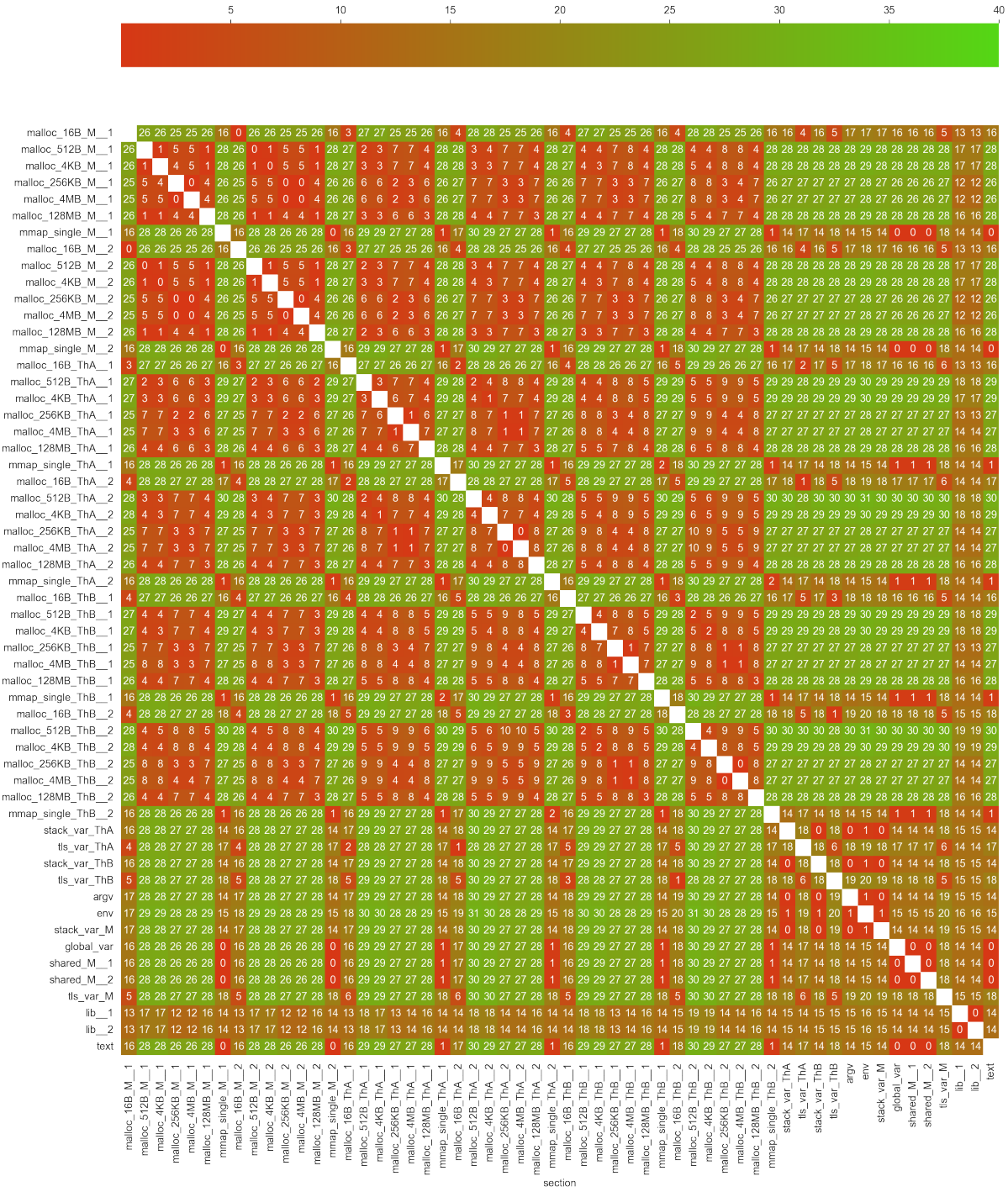

Figure B.5: Correlation Entropy MacOS M1 Rosetta

Object	Ent	Object	Ent
malloc_16B_M__1	-0.087	malloc_16B_ThB__1	0.036
malloc_512B_M__1	-0.054	malloc_512B_ThB__1	-0.068
malloc_4KB_M__1	-0.078	malloc_4KB_ThB__1	0.043
malloc_256KB_M__1	0.000	malloc_256KB_ThB__1	-0.000
malloc_4MB_M__1	-0.013	malloc_4MB_ThB__1	-0.257
malloc_128MB_M__1	-0.012	malloc_128MB_ThB__1	-0.007
mmap_single_M__1	0.000	mmap_single_ThB__1	0.000
malloc_16B_M__2	-0.107	malloc_16B_ThB__2	-0.135
malloc_512B_M__2	-0.060	malloc_512B_ThB__2	-0.236
malloc_4KB_M__2	-0.086	malloc_4KB_ThB__2	-0.335
malloc_256KB_M__2	-0.027	malloc_256KB_ThB__2	-0.439
malloc_4MB_M__2	-0.028	malloc_4MB_ThB__2	-0.456
malloc_128MB_M__2	-0.007	malloc_128MB_ThB__2	-0.012
mmap_single_M__2	0.000	mmap_single_ThB__2	0.000
malloc_16B_ThA__1	0.043	stack_var_ThA	0.000
malloc_512B_ThA__1	-0.027	tls_var_ThA	-0.374
malloc_4KB_ThA__1	0.014	stack_var_ThB	0.000
malloc_256KB_ThA__1	-0.019	tls_var_ThB	-0.213
malloc_4MB_ThA__1	-0.249	argv	-0.387
malloc_128MB_ThA__1	-0.009	env	0.010
mmap_single_ThA__1	0.000	stack_var_M	-0.390
malloc_16B_ThA__2	-0.306	global_var	0.000
malloc_512B_ThA__2	-0.345	shared_M__1	0.000
malloc_4KB_ThA__2	-0.322	shared_M__2	0.000
malloc_256KB_ThA__2	-0.400	tls_var_M	0.028
malloc_4MB_ThA__2	-0.411	lib__1	12.225
malloc_128MB_ThA__2	-0.018	lib__2	12.225
mmap_single_ThA__2	0.000	text	0.000

Table B.4: Absolute Entropy Change MacOS M1 Rosetta reboot

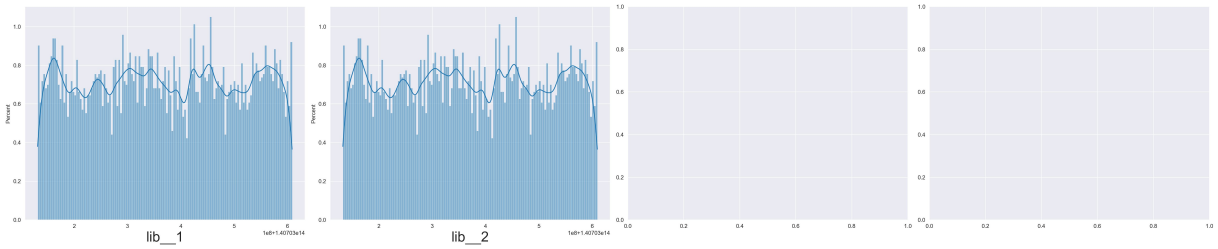

Figure B.6: Probability Distribution MacOS M1 Rosetta reboot new sections

C | Windows Results

Object	Ent	Object	Ent
malloc_16B_M__1	27.408	mmap_single_ThA__2	25.079
malloc_512B_M__1	25.914	malloc_16B_ThB__1	27.408
malloc_4KB_M__1	25.914	malloc_512B_ThB__1	27.408
malloc_256KB_M__1	25.359	malloc_4KB_ThB__1	27.408
malloc_4MB_M__1	27.408	malloc_256KB_ThB__1	26.879
malloc_128MB_M__1	25.528	malloc_4MB_ThB__1	24.656
mmap_huge_M__1	19.654	malloc_128MB_ThB__1	23.153
mmap_single_M__1	25.243	mmap_huge_ThB__1	19.102
malloc_16B_M__2	27.409	mmap_single_ThB__1	25.185
malloc_512B_M__2	27.408	malloc_16B_ThB__2	27.409
malloc_4KB_M__2	27.408	malloc_512B_ThB__2	27.408
malloc_256KB_M__2	27.409	malloc_4KB_ThB__2	27.408
malloc_4MB_M__2	24.122	malloc_256KB_ThB__2	26.749
malloc_128MB_M__2	23.340	malloc_4MB_ThB__2	24.014
mmap_huge_M__2	19.466	malloc_128MB_ThB__2	22.538
mmap_single_M__2	25.209	mmap_huge_ThB__2	18.749
malloc_16B_ThA__1	27.408	mmap_single_ThB__2	25.098
malloc_512B_ThA__1	27.408	stack_var_ThA	27.407
malloc_4KB_ThA__1	27.409	tls_var_ThA	27.409
malloc_256KB_ThA__1	25.881	stack_var_ThB	27.407
malloc_4MB_ThA__1	24.755	tls_var_ThB	27.409
malloc_128MB_ThA__1	23.057	argv	25.416
mmap_huge_ThA__1	19.002	env	25.914
mmap_single_ThA__1	25.158	stack_var_M	27.407
malloc_16B_ThA__2	27.408	global_var	0.000
malloc_512B_ThA__2	27.409	shared_M__1	25.345
malloc_4KB_ThA__2	27.408	shared_M__2	25.345
malloc_256KB_ThA__2	27.407	tls_var_M	25.359
malloc_4MB_ThA__2	23.838	lib__1	0.000
malloc_128MB_ThA__2	22.394	lib__2	0.000
mmap_huge_ThA__2	18.634	text	0.000

Table C.1: Absolute Entropy Windows 11

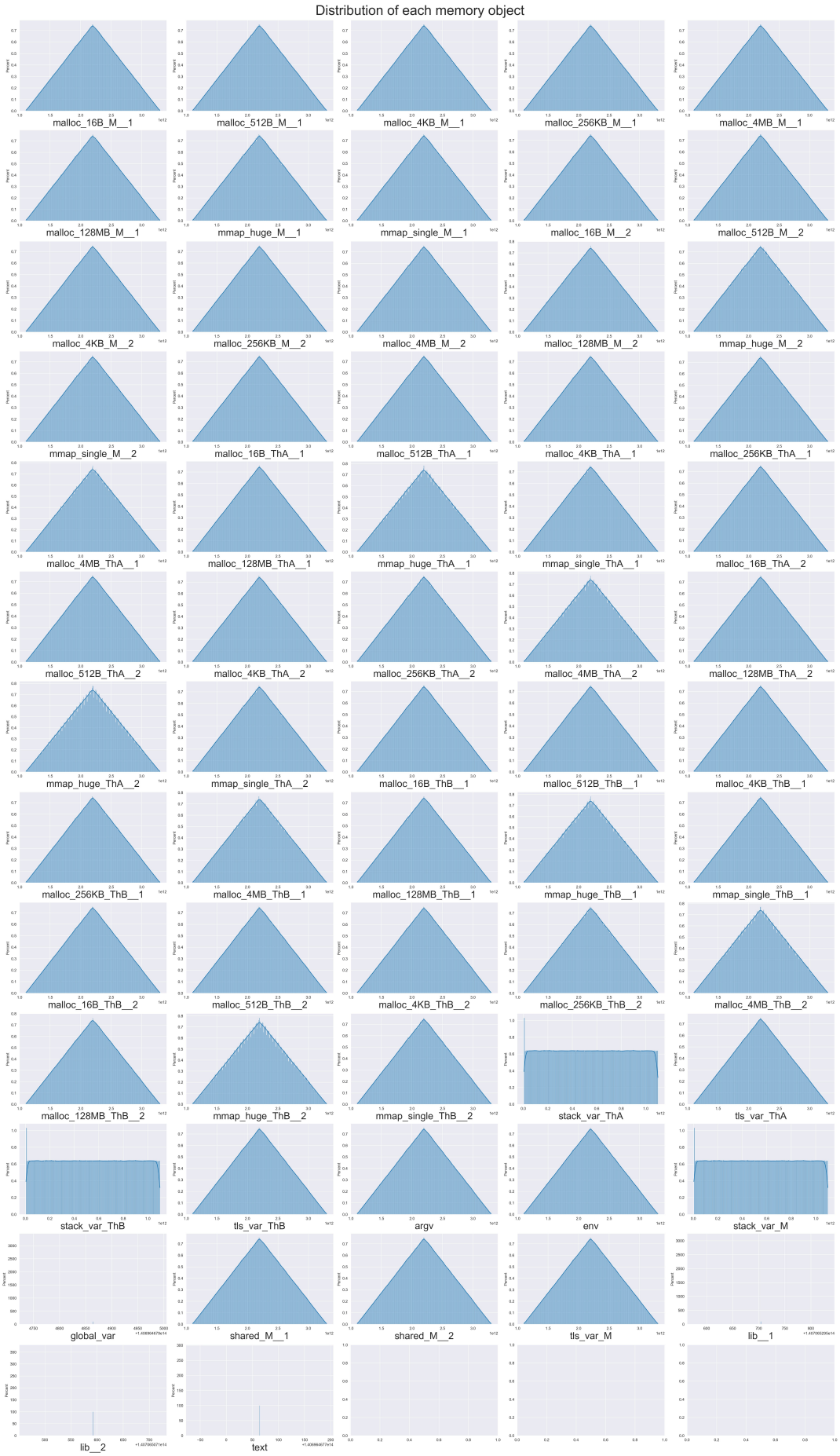

Figure C.1: Probability Distribution Windows 11

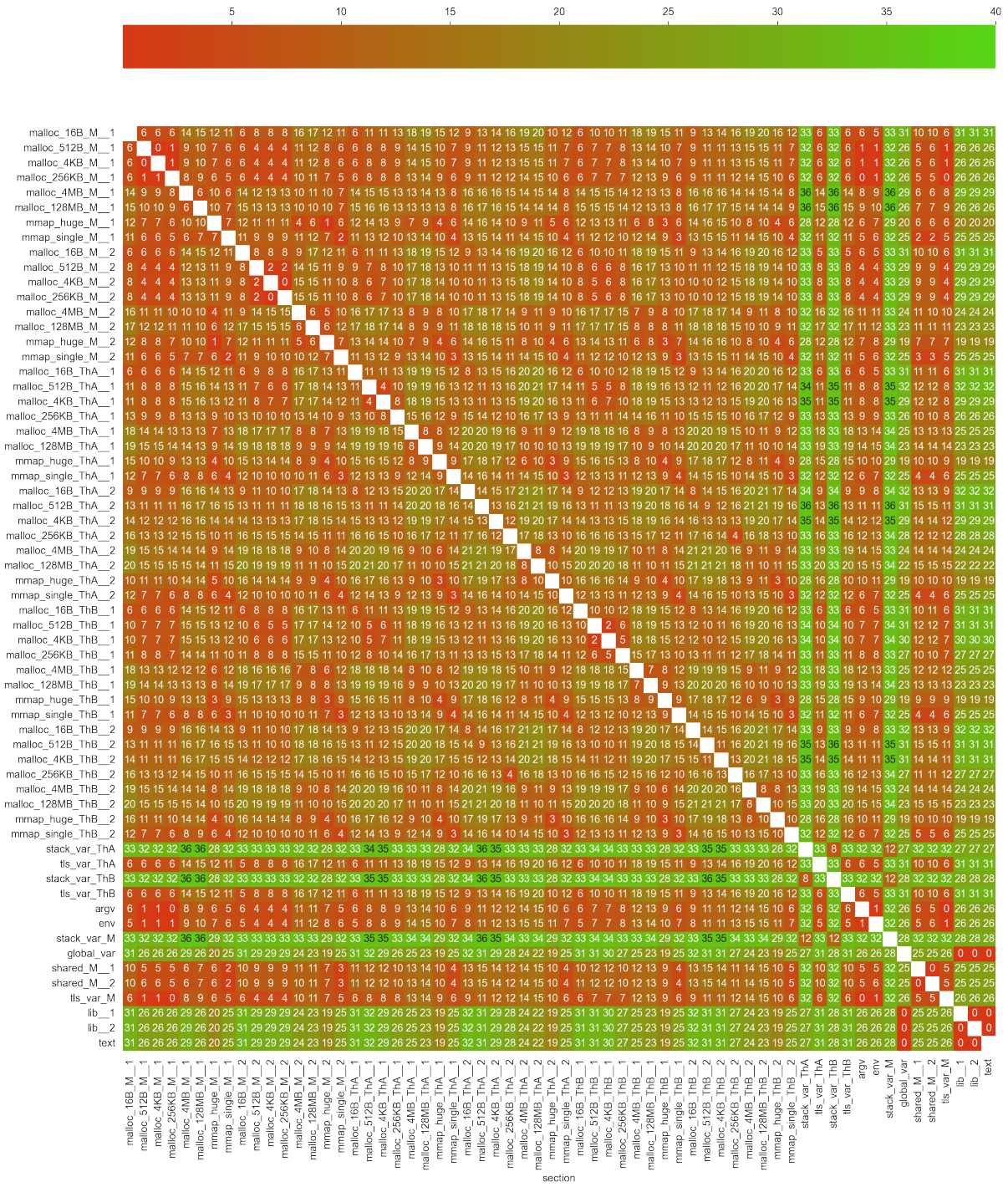

Figure C.2: Correlation Entropy Windows 11

Object	Ent	Object	Ent
malloc_16B_M__1	27.408	mmap_single_ThA__2	25.078
malloc_512B_M__1	25.382	malloc_16B_ThB__1	27.409
malloc_4KB_M__1	25.382	malloc_512B_ThB__1	27.408
malloc_256KB_M__1	25.351	malloc_4KB_ThB__1	27.409
malloc_4MB_M__1	27.409	malloc_256KB_ThB__1	26.918
malloc_128MB_M__1	25.595	malloc_4MB_ThB__1	24.652
mmap_huge_M__1	19.649	malloc_128MB_ThB__1	23.200
mmap_single_M__1	25.240	mmap_huge_ThB__1	19.024
malloc_16B_M__2	27.409	mmap_single_ThB__1	25.183
malloc_512B_M__2	27.372	malloc_16B_ThB__2	27.408
malloc_4KB_M__2	27.408	malloc_512B_ThB__2	27.409
malloc_256KB_M__2	27.408	malloc_4KB_ThB__2	27.409
malloc_4MB_M__2	24.114	malloc_256KB_ThB__2	27.344
malloc_128MB_M__2	23.360	malloc_4MB_ThB__2	23.930
mmap_huge_M__2	19.462	malloc_128MB_ThB__2	22.476
mmap_single_M__2	25.207	mmap_huge_ThB__2	18.607
malloc_16B_ThA__1	27.409	mmap_single_ThB__2	25.084
malloc_512B_ThA__1	27.409	stack_var_ThA	27.407
malloc_4KB_ThA__1	27.408	tls_var_ThA	27.409
malloc_256KB_ThA__1	24.772	stack_var_ThB	27.407
malloc_4MB_ThA__1	24.987	tls_var_ThB	27.408
malloc_128MB_ThA__1	23.138	argv	25.382
mmap_huge_ThA__1	18.997	env	25.382
mmap_single_ThA__1	25.165	stack_var_M	27.408
malloc_16B_ThA__2	27.408	global_var	13.214
malloc_512B_ThA__2	27.408	shared_M__1	25.340
malloc_4KB_ThA__2	27.408	shared_M__2	25.341
malloc_256KB_ThA__2	27.408	tls_var_M	25.382
malloc_4MB_ThA__2	24.036	lib__1	13.250
malloc_128MB_ThA__2	22.452	lib__2	13.273
mmap_huge_ThA__2	18.592	text	13.214

Table C.2: Absolute Entropy Windows 11 reboot

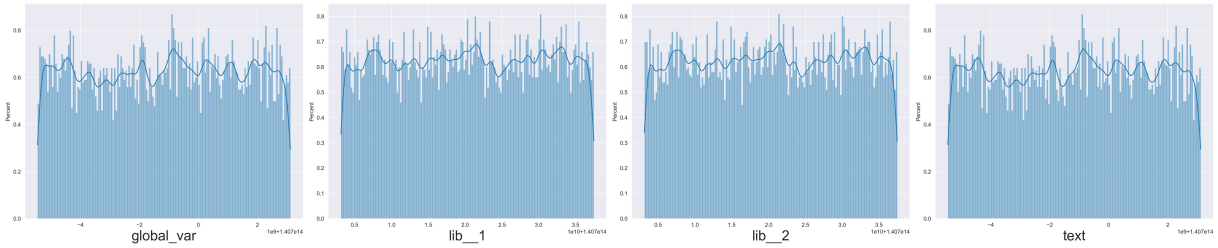

Figure C.3: Probability Distribution Windows 11 reboot new sections

Object	Ent	Object	Ent
malloc_16B_M__1	0.000	mmap_single_ThA__2	-0.001
malloc_512B_M__1	-0.532	malloc_16B_ThB__1	0.001
malloc_4KB_M__1	-0.532	malloc_512B_ThB__1	-0.000
malloc_256KB_M__1	-0.008	malloc_4KB_ThB__1	0.001
malloc_4MB_M__1	0.001	malloc_256KB_ThB__1	0.039
malloc_128MB_M__1	0.067	malloc_4MB_ThB__1	-0.003
mmap_huge_M__1	-0.005	malloc_128MB_ThB__1	0.047
mmap_single_M__1	-0.003	mmap_huge_ThB__1	-0.079
malloc_16B_M__2	0.000	mmap_single_ThB__1	-0.002
malloc_512B_M__2	-0.036	malloc_16B_ThB__2	-0.001
malloc_4KB_M__2	-0.000	malloc_512B_ThB__2	0.001
malloc_256KB_M__2	-0.001	malloc_4KB_ThB__2	0.001
malloc_4MB_M__2	-0.008	malloc_256KB_ThB__2	0.595
malloc_128MB_M__2	0.020	malloc_4MB_ThB__2	-0.084
mmap_huge_M__2	-0.004	malloc_128MB_ThB__2	-0.063
mmap_single_M__2	-0.002	mmap_huge_ThB__2	-0.142
malloc_16B_ThA__1	0.001	mmap_single_ThB__2	-0.014
malloc_512B_ThA__1	0.001	stack_var_ThA	0.000
malloc_4KB_ThA__1	-0.000	tls_var_ThA	0.000
malloc_256KB_ThA__1	-1.109	stack_var_ThB	-0.000
malloc_4MB_ThA__1	0.232	tls_var_ThB	-0.001
malloc_128MB_ThA__1	0.082	argv	-0.033
mmap_huge_ThA__1	-0.005	env	-0.532
mmap_single_ThA__1	0.007	stack_var_M	0.001
malloc_16B_ThA__2	0.000	global_var	13.214
malloc_512B_ThA__2	-0.001	shared_M__1	-0.004
malloc_4KB_ThA__2	0.000	shared_M__2	-0.004
malloc_256KB_ThA__2	0.001	tls_var_M	0.023
malloc_4MB_ThA__2	0.198	lib__1	13.250
malloc_128MB_ThA__2	0.058	lib__2	13.273
mmap_huge_ThA__2	-0.042	text	13.214

Table C.3: Absolute Entropy Change Windows 11 reboot

D | Android Results

Object	Ent	Object	Ent
malloc_16B_M__1	4.320	mmap_single_ThA__2	1.786
malloc_512B_M__1	2.187	malloc_16B_ThB__1	5.490
malloc_4KB_M__1	0.433	malloc_512B_ThB__1	2.990
malloc_256KB_M__1	0.356	malloc_4KB_ThB__1	1.863
malloc_4MB_M__1	7.798	malloc_256KB_ThB__1	1.404
malloc_128MB_M__1	14.006	malloc_4MB_ThB__1	13.283
mmap_single_M__1	0.799	malloc_128MB_ThB__1	15.340
malloc_16B_M__2	4.189	mmap_single_ThB__1	1.767
malloc_512B_M__2	2.219	malloc_16B_ThB__2	5.743
malloc_4KB_M__2	1.067	malloc_512B_ThB__2	2.995
malloc_256KB_M__2	0.036	malloc_4KB_ThB__2	1.493
malloc_4MB_M__2	7.798	malloc_256KB_ThB__2	1.267
malloc_128MB_M__2	14.005	malloc_4MB_ThB__2	15.936
mmap_single_M__2	0.798	malloc_128MB_ThB__2	15.241
malloc_16B_ThA__1	5.336	mmap_single_ThB__2	1.735
malloc_512B_ThA__1	3.109	stack_var_ThA	7.799
malloc_4KB_ThA__1	1.845	tls_var_ThA	5.807
malloc_256KB_ThA__1	1.226	stack_var_ThB	7.791
malloc_4MB_ThA__1	10.632	tls_var_ThB	5.784
malloc_128MB_ThA__1	15.277	argv	0.036
mmap_single_ThA__1	1.743	env	0.000
malloc_16B_ThA__2	5.506	stack_var_M	0.000
malloc_512B_ThA__2	3.121	global_var	13.104
malloc_4KB_ThA__2	1.389	tls_var_M	4.452
malloc_256KB_ThA__2	1.379	lib__1	0.000
malloc_4MB_ThA__2	15.378	lib__2	0.000
malloc_128MB_ThA__2	15.277	text	13.104

Table D.1: Absolute Entropy Android 13

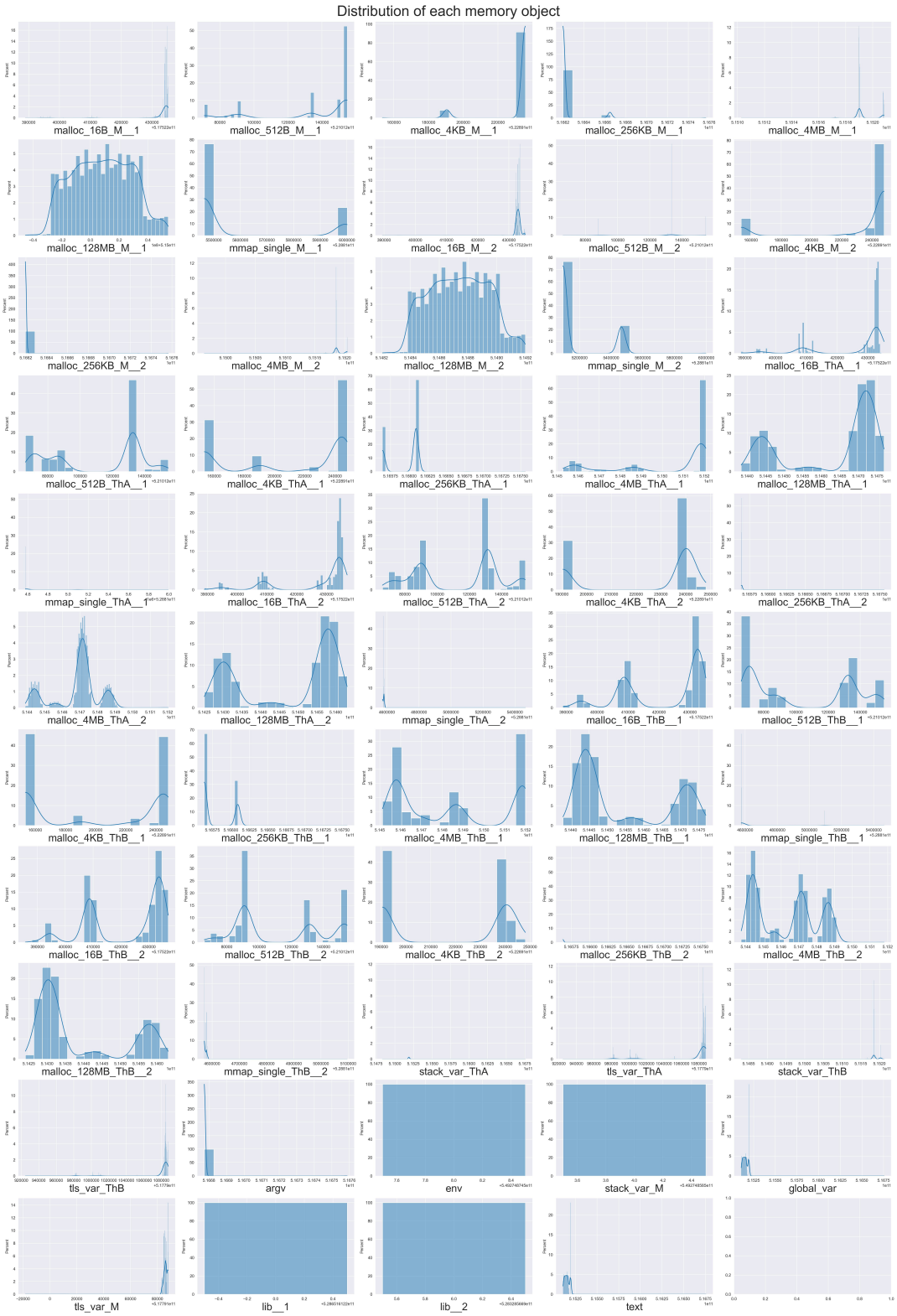

Figure D.1: Probability Distribution Android 13

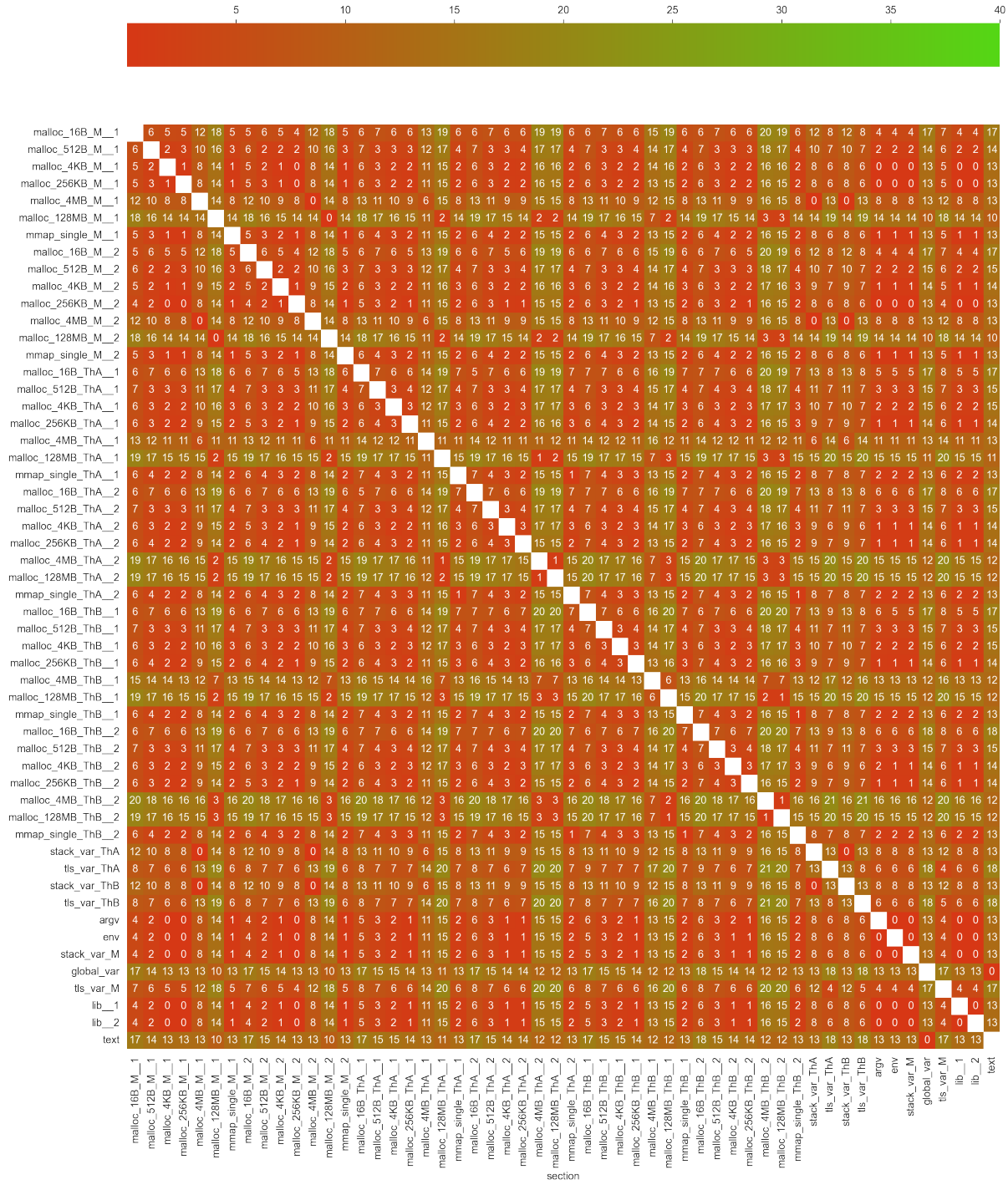

Figure D.2: Correlation Entropy Android 13

Object	Ent	Object	Ent
malloc_16B_M__1	14.459	mmap_single_ThA__2	11.271
malloc_512B_M__1	12.087	malloc_16B_ThB__1	15.666
malloc_4KB_M__1	10.212	malloc_512B_ThB__1	12.956
malloc_256KB_M__1	10.410	malloc_4KB_ThB__1	11.545
malloc_4MB_M__1	16.828	malloc_256KB_ThB__1	11.711
malloc_128MB_M__1	19.742	malloc_4MB_ThB__1	19.176
mmap_single_M__1	10.430	malloc_128MB_ThB__1	19.828
malloc_16B_M__2	14.245	mmap_single_ThB__1	11.241
malloc_512B_M__2	12.087	malloc_16B_ThB__2	15.854
malloc_4KB_M__2	10.689	malloc_512B_ThB__2	12.942
malloc_256KB_M__2	10.489	malloc_4KB_ThB__2	11.210
malloc_4MB_M__2	16.830	malloc_256KB_ThB__2	11.874
malloc_128MB_M__2	19.742	malloc_4MB_ThB__2	19.837
mmap_single_M__2	10.424	malloc_128MB_ThB__2	19.826
malloc_16B_ThA__1	15.448	mmap_single_ThB__2	11.224
malloc_512B_ThA__1	12.973	stack_var_ThA	16.808
malloc_4KB_ThA__1	11.560	tls_var_ThA	15.958
malloc_256KB_ThA__1	11.569	stack_var_ThB	16.789
malloc_4MB_ThA__1	17.945	tls_var_ThB	15.827
malloc_128MB_ThA__1	19.825	argv	10.546
mmap_single_ThA__1	11.254	env	9.830
malloc_16B_ThA__2	15.491	stack_var_M	9.837
malloc_512B_ThA__2	12.962	global_var	18.781
malloc_4KB_ThA__2	11.240	tls_var_M	14.957
malloc_256KB_ThA__2	11.782	lib__1	9.835
malloc_4MB_ThA__2	19.828	lib__2	9.835
malloc_128MB_ThA__2	19.829	text	18.781

Table D.2: Absolute Entropy Android 13 reboot

Object	Ent	Object	Ent
malloc_16B_M__1	10.138	mmap_single_ThA__2	9.485
malloc_512B_M__1	9.900	malloc_16B_ThB__1	10.177
malloc_4KB_M__1	9.780	malloc_512B_ThB__1	9.966
malloc_256KB_M__1	10.054	malloc_4KB_ThB__1	9.682
malloc_4MB_M__1	9.031	malloc_256KB_ThB__1	10.307
malloc_128MB_M__1	5.736	malloc_4MB_ThB__1	5.893
mmap_single_M__1	9.632	malloc_128MB_ThB__1	4.488
malloc_16B_M__2	10.056	mmap_single_ThB__1	9.474
malloc_512B_M__2	9.868	malloc_16B_ThB__2	10.111
malloc_4KB_M__2	9.622	malloc_512B_ThB__2	9.947
malloc_256KB_M__2	10.453	malloc_4KB_ThB__2	9.717
malloc_4MB_M__2	9.032	malloc_256KB_ThB__2	10.607
malloc_128MB_M__2	5.737	malloc_4MB_ThB__2	3.901
mmap_single_M__2	9.625	malloc_128MB_ThB__2	4.585
malloc_16B_ThA__1	10.113	mmap_single_ThB__2	9.489
malloc_512B_ThA__1	9.864	stack_var_ThA	9.009
malloc_4KB_ThA__1	9.715	tls_var_ThA	10.150
malloc_256KB_ThA__1	10.343	stack_var_ThB	8.998
malloc_4MB_ThA__1	7.313	tls_var_ThB	10.042
malloc_128MB_ThA__1	4.548	argv	10.509
mmap_single_ThA__1	9.511	env	9.830
malloc_16B_ThA__2	9.985	stack_var_M	9.837
malloc_512B_ThA__2	9.841	global_var	5.676
malloc_4KB_ThA__2	9.851	tls_var_M	10.505
malloc_256KB_ThA__2	10.403	lib__1	9.835
malloc_4MB_ThA__2	4.450	lib__2	9.835
malloc_128MB_ThA__2	4.553	text	5.676

Table D.3: Absolute Entropy Change Android 13 reboot

List of Figures

List of Tables

List of Symbols

Variable	Description	SI unit
S	estimated entropy	bit
K	number of samples	.

Acknowledgements

I would like to express my sincere gratitude to my advisor Professor Mario Polino for his continuous support, motivation, and knowledge. His guidance helped me throughout my research and writing of this thesis efficiently and successfully. I would also like to sincerely thank my co-advisor Lorenzo Binosi for his insightful assistance and feedback which guided me to shape and enrich this research. This work wouldn't exist without his patient support.
In addition, I wish to acknowledge the entire NECSTLab for fostering a collaborative and intellectually stimulating environment that motivated me throughout this project. I will always be grateful for the knowledge acquired during the time spent with them.