Leveraging Relocations in ELF-Binaries for Linux Kernel Version Identification

By
Manish Bhatt, Irfan Ahmed

From the proceedings of
The Digital Forensic Research Conference
DFRWS 2018 USA
Providence, RI (July 15th - 18th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and forensic challenges to help drive the direction of research and development.

https://dfrws.org
Leveraging relocations in ELF-binaries for Linux kernel version identification

Manish Bhatt, Irfan Ahmed*

Department of Computer Science, University of New Orleans, 2000 Lakeshore Dr., New Orleans, LA, 70122, USA

A B S T R A C T

Identification of operating system kernel version is essential in a large number of forensic and security applications in both cloud and local environments. Prior state-of-the-art uses complex differential analysis of several aspects of kernel implementation and knowledge of kernel data structures. In this paper, we present a working research prototype codeid-elf for ELF binaries based on its Windows counterpart codeid, which can identify kernels through relocation entries extracted from the binaries. We show that relocation-based signatures are unique and distinct and thus, can be used to accurately determine Linux kernel versions and derandomize the base address of the kernel in memory (when kernel Address Space Layout Randomization is enabled). We evaluate the effectiveness of codeid-elf on a subset of Linux kernels and find that the relocations in kernel code have nearly 100% code coverage and low similarity (uniqueness) across various kernels. Finally, we show that codeid-elf, which leverages relocations in kernel code, can detect all kernel versions in the test set with almost 100% page hit rate and nearly zero false negatives.

1. Introduction

Identification of operating system (OS) kernel version is important in both proactive security monitoring and penetration testing, and reactive forensic applications (Ahmed et al., 2015b; Ahmed et al., 2012; Javaid et al., 2012; Ahmed et al., 2015a; Ahmed et al., 2013; Bhatt et al., 2018; Grimm et al., 2017). For example, correct identification of the kernel in a memory snapshot enables a memory forensics toolkit (such as Volatility (Ligh et al., 2014)) to correctly parse important kernel data structures for forensic analysis; network packet-based kernel version identification (such as nmap (Lyon, 2009) and nessus (Lampe, 2005)) enables the penetration tester to select particular exploits pertaining to the specific OS kernel version.

One major problem during kernel version identification is the ability to find the location of the base address of the kernel binary in a memory dump. During the process of booting, kernel base address is randomized every boot using kernel Address Space Layout Randomization (KASLR) (Gross et al., 2017) to protect against exploitation techniques such as return-oriented programming (ROP) (Prandini and Ramilli, 2012). By randomizing the address, the attacker is not able to exploit the system even if they have access to the offsets of certain functions via System.map. Although KASLR was meant to be a defensive technique against exploitation, it presents the forensic investigators with an additional hurdle of figuring out the base address of the kernel before any further investigation can be performed on the memory snapshot of a live system.

Kernel code fingerprinting systems require high coverage of the main kernel code to derive sufficient information to differentiate among various similar kernel versions containing minor differences at binary level (Hebbal et al., 2017). Thus, the signatures obtained from the kernels must be unique or have a low degree of similarity between various versions.

The state-of-the-art techniques in kernel version identification are based on complex differential analysis of several aspects of kernel code implementation (Gu et al., 2012) or complicated analysis of the memory dumps (Gu et al., 2012; Hebbal et al., 2017; Gu et al., 2014). Furthermore, the current techniques for the derandomization of kernel base-address employ brute forcing and information leakage. Realizing the need for a comprehensive solution for Linux kernels, in this paper, we present a technique used to leverage entries in relocatable code tables in kernel elf executables to perform kernel version identification (by detecting kernel pages)
and further derandomize the base address of the kernel. We present a research prototype \texttt{codeid-elf} which uses the relocation entries in the elf kernel binary to create robust signatures and matches them as per an algorithm for both kernel version identification and base address derandomization. \texttt{codeid-elf} is entirely automated as it does not require human intervention in the loop to generate and use the signatures. Finally, we evaluate our implementation of \texttt{codeid-elf} against 22 kernel versions representing a blend of neighbor kernel releases that are likely to have similar code implementations. We find that \texttt{codeid-elf} can accurately detect the versions of kernels with almost 100\% page hit rate.

This paper is organized as follows: it presents the background of Kernel Version Identification and Kernel base address derandomization, then discusses the architecture and methodology of the implementation of \texttt{codeid-elf}. It further presents the evaluation of \texttt{codeid-elf} using many Ubuntu-16.04 LTS 32-bit memory dumps and their corresponding kernel version binaries, and finally concludes along with the scope for future work.

2. Background work

This section mainly discusses the previous work on OS-kernel version identification and derandomization of Base Address of Kernel.

2.1. OS-kernel version identification

Deep analysis techniques are commonly used in the past for kernel version identification. They parse and interpret memory captures to find implementation traits unique to kernel versions.

Gu et al. (2012) propose OS-Sommelier, which extensively analyzes and parses the memory dump for kernel version identification by leveraging virtual to physical address translation and disassembling code. To create a signature of the kernel, the tool takes a snapshot of the memory of the known kernel and processes it in three major steps. First, it identifies the page global directory (PGD) in the dump for virtual-to-physical address translation. Second, it identifies readable pages in kernel code by using the fact that kernel-code pages are marked as read-only for code protection. After the core-kernel code has been identified, the tool generates cryptographic hashes of these kernel-pages after normalization by zeroing out any pointer values in the kernel pages that might have been changed. These hashes are the known kernel signatures. Lastly, a memory dump of an unknown kernel is taken, and steps mentioned above are applied to the dump to create a list of known kernel signatures. To identify kernel versions, a comparison is made between the signatures obtained from a target memory dump and previously identified known signatures. Since cryptographic hashes are involved, it is safe to assume that this process is time-consuming relative to other methods which avoid hashing. Also, disassembly of the kernel code to zero out the pointer values is logically a complicated process.

Lin et al. propose Siggaph (Lin et al., 2011), which relies on identifying in-memory kernel data structures. One main limitation of this approach is that kernel data structures tend to remain the same in many minor kernel releases. Furthermore, data structure changes across different kernel versions are hard to detect and may require a manual reverse engineering effort by the researcher, which increases the intrinsic level of tedium in using this approach in real life.

Volatility (Ligh et al., 2014) is a memory forensic toolkit for forensic analysis of memory images. Knowledge of the precise OS kernel version is important for many of its functions. Volatility implements its kernel version identification for Windows and Linux both by maintaining a profile of a kernel version for parsing memory captures and applying correct set of data-structure definitions. The \texttt{imageinfo} plugin (Ligh et al., 2014) in volatility scans a whole memory dump using predetermined signatures to find the kernel debugging symbols table which contains the exact kernel version information. However, this approach is fragile because the signature-dependent values are vulnerable to malicious modifications. Furthermore it has considerable overhead because the process of deriving signatures is not fully automated.

Ahmed et al. (2015b) propose \texttt{codeid}, which generates signatures from relocation entries in the executables of the Portable Executable (PE) format for various programs and uses a differential technique during the matching process. The generated signatures consist of an \texttt{(offset, pointer)} tuple. \texttt{Offset} refers to the offset of the relocation location with reference to the base address of the executable, whereas \texttt{pointer} refers to the 32- or 64-bit value of the pointer located at that particular offset. The signatures are said to match if the difference of the \texttt{pointer} and the base address of the code in memory is the same as the difference of the \texttt{pointer} and the base address of the code in the PE executable for all the signatures. However, for different relocation entries, the difference of the values can be different. Also, this technique expects a secondary loop to be run to derandomize the kernel. In addition, \texttt{codeid} is designed and evaluated for Windows kernels.

Hebbal et al. (Hebbal et al., 2017) propose another method called \texttt{k-BinID} that locates, fingerprints, and derandomizes operating system kernels at run time by using static binary analysis. They also introduce a technique called \texttt{backward disassembly} to correctly locate instructions that precede a given virtual address in the memory. The main drawback of this method is that \texttt{k-binID} relies heavily on a database of kernel code blocks signatures in order to precisely fingerprint the main kernel binary code. Thus, if the said signatures are unavailable, this method is not applicable.

Network fingerprinting tools such as \texttt{nmap} (Lyon, 2009) and \texttt{Xprobe2} (Arkin et al., 2003) remotely identify the kernel version via analysis of the network packets being exchanged. Since the implementation of the TCP/IP stack is different in different OSes, the intrinsic differences between the crafted packets can be used to identify the OS versions. However, we are looking to identify the kernel versions in memory dumps as opposed to live systems. Such network-based approaches for OS-identification are beyond the scope of our work.

There are several other approaches which leverage the interrupt descriptor table (IDT) and global descriptor table (GDT) and the ways these tables are set up in different OSes and architectures. For instance, Christodorescu et al. (Christodorescu et al., 2009) compute cryptographic hashes of interrupt handler code, and then use them as signatures to identify different kernels. Quynh et al. (Quynh, 2018) on the other hand propose kernel version identification approach by leveraging the fact that the protected mode of the Intel platform enforces few constraints on kernel implementation.

2.2. Derandomizing Base Address of Kernel

As mentioned, derandomizing the base address of the kernel is one of the major hurdles for the forensic analysis of a memory dump. Several attempts at breaking user space ASLR have been made over the years. Shacham et al. (2004) demonstrate a simple brute force approach, which requires only 2^{16} probes to derandomize a vulnerable 32-bit program. Information leakage is another way to break ASLR. Furthermore, Jang et al. use Intel TSX to derandomize Linux Kernels (Jang et al., 2016). Hund et al. leverage practical hardware level side-channel attacks against KASLR (Hund et al., 2013).
Volatility derandomizes the kernel base address by identifying KdDebuggerDataBlock (KDBG) (bneuburg, 2017; Li et al., 2014), a data structure maintained by the Windows kernel for debugging purposes in memory. For Linux systems, Volatility looks for the string "swapper" followed by a few null bytes in memory, checks certain conditions to see if the location of the string is relevant or not, and finally determines the KASLR shift by subtracting physical address of init_task from the physical address where said conditions apply. It is possible for malware to mimic the nature of the "swapper" string (bneuburg, 2017) and confuse Volatility, which makes this approach somewhat unreliable.

Gu and Lin (2016) present four approaches to breaking KASLR for forensic applications, viz., patched-code based approach, unpatched-code based approach, brute-force approach, and read-only pointer based approach. These approaches, however, have high performance overhead and depend on a deep understanding of kernel for derandomization. For instance, they require brute-forcing, comprehensive parsing of elf-binary, disassembling of kernel code, finding data structures in memory such as Global Descriptor Table (GDT), etc.

3. Design rationale

This section presents an overview of codeid-elf and various details associated with it.

3.1. Design Technique

Let \( S \) be one of the obtained signatures. \( S \) is a tuple consisting of an offset from the base address and the pointer value at that offset in disk. Let \( P(i) \) refer to the \( i \)th page in memory. Also, let \( \text{ptr} \) be the value at \( P(i)+S.\text{offset} \) in memory. So, if \( \text{ptr} \) belongs to the kernel code which starts at the base address \( B_m \) and the code in disk starts at the base address \( D_m \), then the following is true:

\[
B_m - D_m = \text{RandomizedOffset} = \text{ptr} - S.\text{pointer}
\]  

(1)

If RandomOffset is the same across several or all signatures in the page, then the page matches the executable from which the signatures are extracted. Also, if all the pages in the .text section of the Linux (ELF format) executable match, then the execution of the executable has been detected in memory by codeid-elf.

The following can be inferred from equation (1), RandomOffset can be zero if ASLR is not enabled. It can be less than \( B_m \) if an executable is loaded at a lower memory location than \( D_m \), the base address of the code on disk. This will not be the case as in Linux as the kernel is loaded at or above virtual addresses starting at \( 0xC0000000 \). In general, it should also be noted that if paging is enabled, then only certain pages of the executable might be detected from memory. Moreover, if the pages that are detected are not found in the Global Descriptor table (GDT), then remnants of the code detection from a previous code execution is detected.

**Data:** output of readelf -r program  
**Result:** List of \( \langle \text{offset}, \text{pointer} \rangle \) signature tuples initialization;

\[
\text{filehandler} = \text{Open Output File} ;
\]

\[
\text{while every line in the output do}
\]

\[
\text{if (line contains "R_386.32") then}
\]

\[
\text{filehandler.write(line.offset+" "+line.value) ;}
\]

\[
\text{else}
\]

\[
\text{continue ;}
\]

end

end

**Algorithm 1.** Algorithm to extract signatures.

Figs. 1 and 2 illustrates this concept with reference to the kernel version 4.10.0-28-generic. Here, we see that the difference between the in-memory code pointer value and the in-disk code pointer value corresponds to the randomized offset. If this difference is the same across all signatures for the page, then the page is said to have matched the executable.

4. Implementation details

Conceptually, the functionality of codeid-elf can be broken down into following parts:

4.1. Signature generation

Signatures derived from the ELF binary consist of a tuple \( \langle \text{offset}, \text{pointer} \rangle \), where the \text{offset} refers to the location of relocation with reference to the base address of the code segment in memory and/or file and the \text{pointer} refers to the address pointer located at the

<table>
<thead>
<tr>
<th>Relocation Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actual Randomised Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base in Memory - Base on Disk = 0xdb001000 - 0xc1001000 = 0x1a000000</td>
</tr>
</tbody>
</table>

**Fig. 1.** An illustration for the Design Technique of codeid-elf.
said location. The signatures that are derived can then be collected into page-wise groups.

The primary relocation entry type we are interested in is R_386_32. In this type of relocation entry, the pointer value is directly replaced at the said offset during runtime, and hence no modification needs to be done before the offset-pointer tuple is used as a signature. In another type of relocation entry, called R_386_P32, the pointer at the particular offset is the program counter/instruction pointer (IP) relative value and not the actual pointer. Since we do not have the IP value, it is not possible for us to use class of relocation entries. The offsets are extracted from the ELF binary using the readelf -r command line utility. Then, a python script is written to extract the offsets of the relocation entry from the output of readelf utility such that they are of the type R_386_32 as per Algorithm 1. A dictionary indexed using the page address is created out of these offsets.

4.2. Page detection

In this stage in the execution of codeid-elf, the program iterates through all the pages in the memory dump per code-page in the code section and checks for uniformity in the difference between the majority of the pointer values as per Algorithm 2.

4.3. Executable detection

If all pages in the .text segment of the kernel executable are detected according to Algorithm 3, then it is ascertained that the executable has been detected by codeid-elf.

Furthermore, for derandomization of the base address, it is not necessary to run this code for all pages. In this case, just looking for the first page of the kernel code should suffice.

5. Gathering evaluation data

The functionality of codeid-elf was tested against several kernel versions. All experiments were conducted on Intel Core i7-7000 CPU 64-bit with 32 GB of RAM. To conduct experiments, we needed to gather several memory dumps corresponding to various kernel versions, and gather their corresponding vmlinux executables.

5.1. Installing kernel versions

We use a Virtual Machine (VM) to obtain vmlinux executables for the kernels by installing different kernels inside Ubuntu-VM along with their corresponding packages using the apt-get utility.

This enabled us to move towards Memory Snapshot Collection.

5.2. Obtaining memory snapshots

The memory snapshots required for evaluation were obtained from a VM in VMware Workstation 14.0 running Ubuntu 16.04 LTS. To create the memory snapshots, first a snapshot was created in the VM. Then, the *.vmsn and *.vmem files corresponding to the particular snapshot were copied to a familiar location. Then, vmss2core utility was used as follows:

vmss2core -M (+.vmsnfile) (+.vmemfile)

Using vmss2core utility in the manner mentioned above created a file called vms2_core which contained physical layout of the guest-VM memory. The memory snapshots we used were 512 MB in size (131,072 pages).

5.3. Obtaining vmlinux executable

A linux system does not keep vmlinux executable by default. To obtain the executable from the VM corresponding to the particular kernel, we need to first add the debs repository. To do this, we execute the following commands:

Listing 1. Bash script to obtain vmlinux executable

```
#Add debs repository
echo "deb http://ddeb.ubuntu.com/$(lsb_release -cs)\nupdates\nmain\nrestricted\nuniverse\nmultiverse" \n|> $debs\n
deb http://ddeb.ubuntu.com/$(lsb_release -cs)\nsecurity\nmain\nrestricted\nuniverse\nmultiverse

deb http://ddeb.ubuntu.com/$(lsb_release -cs)\n\nproposed\nmain\nrestricted\nuniverse\nmultiverse"
```

Data: List of Page Signatures, Memory Dump
Result: Detected/Not Detected initialization;
pageaddress = 0;

```
while Page in Memory Dump do
    while signature in List of Page Signatures of type R_386_32 do
        if (Page[signature.offset] - Code in disk
            [signature.offset]) not equals
            (Page[signature.offset+1] - Code in disk
            [signature.offset+1]) then
            matched = false;
        else
            matched = true;
            pageaddress = Page & 0xFFFFFFFF;
        end
    end
end
return (pageaddress, matched);
```

Algorithm 2. Algorithm to detect pages in memory using relocation entries.
Data: Page Groups, Memory Dump
Result: Detected/Not Detected

    page_address = 0;
    while Page Group in Page Groups do
        if algorithm2 PageGroup, MemoryDump then
            print algorithm2 page_address;
        else
            continue;
        end
    return matched;

Algorithm 3. Algorithm to detect all pages of an executable in memory.

    sudo tee -a /etc/apt/sources.list.d/ddebs.list
    sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 428D07C0
    sudo apt-get update
    sudo apt-get install linux-image-$(uname -r)-dbgasm

After script 1 is executed inside the host VM, the vmlinux file for the corresponding kernel version can be found inside /usr/lib/debug/boot/ folder. The ELF-executables found here are copied to the host machine for further analysis. This approach avoids compiling all the kernels from source code. However, the list of linux-image-* packages available on Ubuntu 16.04 LTS was not an exhaustive list of all Linux kernels. All in all, we evaluated codeid-elf using 22 different kernel versions ranging from 4.4.0-24 to 4.10.0-14.

6. Evaluation results

6.1. Relocation prevalence & code coverage

We first study the prevalence and the coverage of the relocations. Prevalence of relocation entries is defined as the average number of relocation entries in the executable per page, whereas coverage of relocation entries is defined as the fraction of pages that have relocation entries in them.

Table 1 presents a summary of our findings. In all cases, we found the minimum number of relocation entries to be two to build a page signature. Moreover, the third column in Table 1 represents (1 - codecoverage). It is the ratio of the pages that have no relocations. Thus, codeid-elf cannot detect these pages. The column shows that such pages are not in significant number and thus, the signatures extracted from relocation entries of kernel executables approximately cover all of the kernel code.

6.2. Accuracy analysis

In this section, we study the effectiveness of codeid-elf in detecting pages belonging to the kernel executable, correctly identifying the base address of the kernel, and also correctly identifying the version of the kernel via fingerprints. To establish accuracy, first we needed to find a way of establishing ground truth.

Establishing Ground Truth. To establish a base for comparison, we used in-VM tools that were available to us. For instance, to find out the base address at which the kernel is loaded, we looked at the address for startup_32 symbol inside /proc/kallsyms file available to us. To find the range of memory in which the kernel code is loaded, we looked at the output of /proc/iomem which showed us the physical address of the base and the end of the kernel. Moreover, we knew the version of the kernel when we took the memory snapshot of that particular kernel using VMware Snapshot Mechanism. We used this previous knowledge to verify if the kernel version was identified correctly or not.

Identifying the base address of the kernel code and the range of the kernel code allowed us to define the following outcomes of experimentation:

- **True Positive (TP):** This refers to the number of pages that have been detected by codeid-elf and actually contain kernel code. It is equivalent to a page hit.
- **False Positive (FP):** This refers to the number of pages that have not been detected by codeid-elf but do not belong to the kernel. It is equivalent to a page false hit.
- **True Negative (TN):** This refers to the number of pages that have not been detected by codeid-elf and actually do not belong to the kernel. It is equivalent to correct rejection.
- **False Negative (FN):** This refers to the number of pages that have not been detected by codeid-elf and actually contain kernel code. It is equivalent to a miss.

6.3. Page detection rate

With the above mentioned outcomes of experiments, we can define the following metrics for page-level accuracy.

\[
\text{Sensitivity} = \frac{TP}{TP + FN} \quad (2)
\]

Table 2 shows the hit rate for various kernel versions. Here, we see that the Sensitivity is very close to 100%. Since kernel code is not paged, we had expected the hit rate to be 100%. During the detection process, only the pages that did not have any signatures were the majority of false negatives. Moreover, we believe that since at runtime, the kernel is able to modify itself via the alternative instructions, this might have resulted in few pages being placed in the false negatives.

In the beginning, we chose to stop when codeid-elf detected the first memory-page that completely matched the respective code-page. However, this resulted in a significant number of false positives. Hence, we opted to iterate through the entire memory dump per single code-page, detect all memory pages with minimum of two signature matches, and pick the memory page with the highest number of signature-matches as the detected in-memory page. This measure reduced our false positives to almost zero. On observation, it was found that the remaining false positives were pages that were in either .data segment or the .bss Segment of the kernel. We had previously obtained the range of kernel code via the /proc/iomem file inside the VM during the process of creating memory dumps. This was a precautionary measure as we did not want codeid-elf to be looking anything but the kernel code. This enabled us to ignore any detected-page that was not in the kernel code range for the particular kernel. These measures reduced our false positive rate to zero.

6.4. Executable-level accuracy

Regarding file-level accuracy, it was first vital to study about the similarity of kernel executable signatures. To do this, we extracted
the signatures from all kernel executables and placed them in sets. Finally, we figured out the cardinality of the intersection of these sets to come up with the number of signatures that were common in two or more kernels.

If \( S \) be the number of common signatures in two kernels \( A \) and \( B \), \( n(A) \) be the number of signatures generated from kernel \( A \), \( n(B) \) be the number of signatures generated from kernel \( B \), and \( \text{Sim}_{AB} \) be the similarity of kernels \( A \) and \( B \) then we calculated similarity as follows:

\[
\text{Sim}_{AB} = \frac{S}{n(A)}
\]

(3)

Conversely, we calculate similarity of kernel \( B \) to \( A \) as follows:

\[
\text{Sim}_{BA} = \frac{S}{n(B)}
\]

(4)

We used Algorithm 4 to calculate similarity between two kernels.

Data: Kernel A and Kernel B ELF executables and list of offsets

Result: Similarity between kernel A and B initialization;

\[
\text{kernel.A.Signatures} = \text{set()};
\]

\[
\text{kernel.B.Signatures} = \text{set()};
\]

\[
\text{intersection.set} = \text{set()};
\]

while Offset in Offsets A do

\[
\text{Kernel.A.Executable.gotot[offset]};
\]

\[
\text{kernel.A.Signatures.append(str(offset) + str(hex(Kernel.A.Executable.read(4))));}
\]

end

while Offset in Offsets B do

\[
\text{Kernel.B.Executable.gotot[offset]};
\]

\[
\text{kernel.B.Signatures.append(str(offset) + str(hex(Kernel.B.Executable.read(4))));}
\]

end

\[
\text{intersection.set} = \text{kernel.A.Signatures} \& \text{kernel.B.Signatures};
\]

\[
\text{Sim}_{AB} = \text{len(intersection.set)}/\text{len(kernel.A.Signatures};
\]

\[
\text{Sim}_{BA} = \text{len(intersection.set)}/\text{len(kernel.B.Signatures};
\]

return (Sim_{AB}, Sim_{BA});

Algorithm 4. Algorithm to detect similarity between two pages.

Table 3 presents similarity between a subset of the distinct kernels in terms of the number of common signatures that were generated by using codeid-elf. As we see, the percent overlap of the generated signatures from the kernel executables is quite low in majority of the cases. This implies that majority of the pages thus detected will be characterized by a set of signatures unique to the particular kernel. This indirectly supports the premise that relocation entries almost uniquely fingerprint kernel, and thus allow for kernel identification.

Keeping this in mind, we evaluated codeid-elf for kernel version identification of the kernels mentioned in Table 2. Codeid-elf was correctly able to identify specific versions of the said kernels with 100% accuracy.

6.5. Performance analysis

The algorithm for codeid-elf extracts signatures from all the pages of the kernel code and compares those signatures against each page in the memory snapshot. Moreover, it was important that signatures pertaining to each page of kernel code be checked against every single page in memory to avoid false positives. However, this meant that we needed to sacrifice some of the performance of codeid-elf.

As opposed to the execution times of similar algorithms (Gu and Lin, 2016; Hebbal et al., 2017), the execution of codeid-elf is several times slower. The execution time statistics for codeid-elf are shown in Fig. 3.

Fig. 3 represents the time taken by codeid-elf for the detection of particular kernel version. We see that execution time for codeid-elf is higher in newer kernels as opposed to older kernels. Firstly, the average execution time is probably high because of codeid-elf having to check every single code-page against all memory-pages. Moreover, the increase in the timing measurements in newer kernels as opposed to older kernels is probably because of increasing number of relocation entries in newer kernels as opposed to older kernels. Also, our implementation of codeid-elf consisted of a mixture of C, Python, and Bash scripts with a significant amount of file IO operations to avoid overwhelming the memory of our host system. This, coupled with an increased number of relocation entries in newer kernels as opposed to older kernels, could possibly have caused the performance of codeid-elf to have deteriorated in newer kernels.

6.6. Possible optimization measures

It should be stressed that our current implementation has not been fully optimized for performance. To increase performance, the following measures could be implemented:

- It is known that Linux Kernels are not paginated, and are located in virtual addresses greater than \texttt{0x00000000} in 32-bit systems. Moreover, finding the base address of the kernel code is a simpler problem as compared to Kernel Version Identification as not all pages have to be detected in order to find it. Since kernel code is stored in a contiguous block, we could incorporate facts about the kernel ranges into codeid-elf to increase its performance.

- As we see in Table 1, the prevalence of relocation entries is quite high. Currently in codeid-elf, we use all relocation entries of the type \texttt{R_386_32} under .rel.text as signatures. We could devise a method to intelligently create a subset of signatures by decreasing the number of signatures without sacrificing code coverage and decreasing relocation prevalence. This would be a major optimization measure that could be applied to the current implementation.

- Implementing codeid-elf entirely in C and removing Python and Bash scripts would also be an optimization measure to boost its performance.

7. Comparison with previous works

The process of identifying kernel version involves devising unique methodology to fingerprint the kernel. In this aspect, we believe that our method is superior as opposed to methods proposed by Hebbal et al. (Hebbal et al., 2017) and (Roussev et al., 2014). A comparison between codeid-elf and previous works based on their reports has been outlined in Table 4. Hebbal et al. reported over 99% similarity in certain code blocks in their study based on k-BiniD. Also, Roussev et al. (Roussev et al., 2014) reported that some of the kernels they did their experiments on had as low as seven unique blocks because changes in code between said kernels was minimal. As Table 3 shows us, the number of same relocation entry-based signatures generated by codeid-elf is quite minimal. Except for kernels 4.11.0-14-generic and 4.11.0-13-generic which have very similar relocations, remaining comparisons revealed that...
our signatures are at least 90% unique. This is excellent, as unique fingerprints translate to higher detection accuracy.

Another advantage of our method as opposed to previous methods is that our method is simple to implement and doesn’t require any extraneous knowledge like disassembly of code (Hebbal et al., 2017; Gu et al., 2012), or complicated memory dump analysis (Gu et al., 2012). It just involves the comparison of the differences between pointer values at specific locations in a page, which makes it easy to comprehend and to implement.

Although we did not do an exhaustive study for all Linux kernels, we believe that our method will work for other kernel versions, and even custom written kernels as long as sufficient amount of relocation entries similar to the type \(R_{386,32}\) are available and the derived signatures have nearly 100% code coverage.

8. Conclusion and future works

In this work, we considered the problem of finding known kernel code in memory dumps for Linux kernels. As a solution, we used relocation entries extracted from the ELF executables and used them as robust unique fingerprints to identify the kernel. We also did a similarity study across 22 kernel binaries with respect to their relocation entries which provided some proof as to the previous claim. Moreover, we showed that although the base address

---

**Table 1**

<table>
<thead>
<tr>
<th>Relocation prevalence and code coverage.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vmlinux Kernel Version</td>
<td>Relocation Prevalence</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>4.4.0–22</td>
<td>55.4482</td>
</tr>
<tr>
<td>4.4.0–24</td>
<td>55.4314</td>
</tr>
<tr>
<td>4.10.0–14</td>
<td>54.0464</td>
</tr>
<tr>
<td>4.10.0–14-lowlatency</td>
<td>53.2292</td>
</tr>
<tr>
<td>4.10.0–19</td>
<td>54.0825</td>
</tr>
<tr>
<td>4.10.0–19-lowlatency</td>
<td>53.2381</td>
</tr>
<tr>
<td>4.10.0–20</td>
<td>54.0825</td>
</tr>
<tr>
<td>4.10.0–21</td>
<td>53.9475</td>
</tr>
<tr>
<td>4.10.0–22</td>
<td>53.9418</td>
</tr>
<tr>
<td>4.10.0–24</td>
<td>53.9442</td>
</tr>
<tr>
<td>4.10.0–26</td>
<td>53.9584</td>
</tr>
<tr>
<td>4.10.0–27</td>
<td>53.9584</td>
</tr>
<tr>
<td>4.10.0–28</td>
<td>53.9584</td>
</tr>
<tr>
<td>4.10.0–30</td>
<td>53.9593</td>
</tr>
<tr>
<td>4.11.0–13</td>
<td>52.9995</td>
</tr>
<tr>
<td>4.11.0–14</td>
<td>52.9995</td>
</tr>
<tr>
<td>4.13.0–16</td>
<td>52.9387</td>
</tr>
<tr>
<td>4.13.0–17</td>
<td>52.9114</td>
</tr>
<tr>
<td>4.13.0–19</td>
<td>52.9171</td>
</tr>
<tr>
<td>4.13.0–20</td>
<td>52.9171</td>
</tr>
<tr>
<td>4.13.0–21</td>
<td>52.8611</td>
</tr>
<tr>
<td>4.13.0–24</td>
<td>52.8611</td>
</tr>
<tr>
<td>4.13.0–25</td>
<td>52.8611</td>
</tr>
</tbody>
</table>

**Table 2**

<table>
<thead>
<tr>
<th>Vmlinux Kernel Version</th>
<th>Number of Generated Signatures</th>
<th>Size of Kernel Code in MB</th>
<th>Total Number of Pages in code</th>
<th>Page Hit Rate</th>
<th>Base Address of Kernel</th>
<th>Derandomisation Kernel Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.0–22</td>
<td>108,734</td>
<td>7.6640</td>
<td>1961</td>
<td>99.47</td>
<td>0xc1000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.4.0–24</td>
<td>108,812</td>
<td>7.6708</td>
<td>1963</td>
<td>99.43</td>
<td>0xc1000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–14</td>
<td>113,584</td>
<td>8.2387</td>
<td>2109</td>
<td>99.52</td>
<td>0xdc000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–14-lowlatency</td>
<td>113,059</td>
<td>8.2976</td>
<td>2124</td>
<td>99.48</td>
<td>0xc6000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–19</td>
<td>114,060</td>
<td>8.2419</td>
<td>2109</td>
<td>99.57</td>
<td>0xc3000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–19-lowlatency</td>
<td>113,131</td>
<td>8.3010</td>
<td>2125</td>
<td>99.38</td>
<td>0xc8000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–20</td>
<td>114,060</td>
<td>8.2419</td>
<td>2109</td>
<td>99.57</td>
<td>0xc1000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–21</td>
<td>114,207</td>
<td>8.2701</td>
<td>2117</td>
<td>99.29</td>
<td>0xed000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–22</td>
<td>114,195</td>
<td>8.2702</td>
<td>2117</td>
<td>99.48</td>
<td>0xce000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–24</td>
<td>114,200</td>
<td>8.2704</td>
<td>2117</td>
<td>99.57</td>
<td>0xc9000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–26</td>
<td>114,230</td>
<td>8.2707</td>
<td>2117</td>
<td>99.62</td>
<td>0xc8000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–27</td>
<td>114,230</td>
<td>8.2707</td>
<td>2117</td>
<td>99.62</td>
<td>0xda000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–28</td>
<td>114,230</td>
<td>8.2708</td>
<td>2117</td>
<td>99.62</td>
<td>0xdb000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.10.0–30</td>
<td>114,232</td>
<td>8.2713</td>
<td>2117</td>
<td>99.43</td>
<td>0xca000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.11.0–13</td>
<td>113,525</td>
<td>8.3678</td>
<td>2142</td>
<td>99.06</td>
<td>0xcd000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.11.0–14</td>
<td>113,525</td>
<td>8.3680</td>
<td>2142</td>
<td>99.06</td>
<td>0xda000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.13.0–13</td>
<td>112,336</td>
<td>8.2896</td>
<td>2122</td>
<td>99.24</td>
<td>0xda200000</td>
<td>√√</td>
</tr>
<tr>
<td>4.13.0–17</td>
<td>112,384</td>
<td>8.2979</td>
<td>2124</td>
<td>99.15</td>
<td>0xcf000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.13.0–19</td>
<td>112,396</td>
<td>8.2987</td>
<td>2124</td>
<td>99.05</td>
<td>0xe000000</td>
<td>√√</td>
</tr>
<tr>
<td>4.13.0–21</td>
<td>112,396</td>
<td>8.2987</td>
<td>2124</td>
<td>99.05</td>
<td>0xc300000</td>
<td>√√</td>
</tr>
<tr>
<td>4.13.0–24</td>
<td>112,277</td>
<td>8.2987</td>
<td>2124</td>
<td>99.24</td>
<td>0xe600000</td>
<td>√√</td>
</tr>
<tr>
<td>4.13.0–25</td>
<td>112,277</td>
<td>8.2987</td>
<td>2124</td>
<td>99.24</td>
<td>0xc400000</td>
<td>√√</td>
</tr>
</tbody>
</table>
of the kernel code could possibly be skewed because of KASLR, we could use simple differential technique to derandomize the kernel as well.

The fundamental merit of this method is that it is quite simple and doesn't require any form of hashing, and/or deep analysis techniques which can result in fragile, non-extendable methods. Although we evaluated this approach to work for Linux kernel version identification and derandomization, it can also easily be extended to other executables. Another advantage is that the only input required to generate signature in this method is the ELF binary, and no knowledge of kernel data structures or analysis of memory dumps is required.

Our experimental evaluation showed that we are able to detect pages belonging to a particular kernel binary with nearly zero misses. Most of the misses were because the corresponding kernel page did not have any relocation entry. Codeid-elf correctly identified memory dumps containing 22 different Linux kernels with a page hit rate of over 99%. This showed that our approach works exceedingly well even in distinguishing kernel versions which are close to one another in terms of their implementations.

The main limiting factor in our method is the execution time of the algorithms. Because signatures belonging to each page of kernel code has to be checked against every single page of memory to avoid false positives, execution of the algorithms for kernel version detection took several minutes. However, we noted that several optimizations could be made to the current implementation of codeid-elf to boost its performance.

In future, we would like to test this approach on custom built Linux kernels. We would also like to extend this approach to test on user-space binaries like Chrome, Firefox etc. Currently we evaluated codeid-elf on 32-bit systems with Intel architecture. We would like to extend it to other architectures like ARM. Finally, although this method isn't targeted towards malware detection, we believe that this approach can be reliably used for ad-hoc malware detection.

### Table 3

**Similarity of Kernels based on codeid-elf signatures.**

<table>
<thead>
<tr>
<th>Kernel A</th>
<th>Kernel B</th>
<th>Number of Common Signatures</th>
<th>SimAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10.0–21-generic</td>
<td>4.10.0–28-generic</td>
<td>23</td>
<td>0.0201</td>
</tr>
<tr>
<td>4.10.0–21-generic</td>
<td>4.10.0–22-generic</td>
<td>2107</td>
<td>1.8449</td>
</tr>
<tr>
<td>4.10.0–21-generic</td>
<td>4.10.0–14-generic</td>
<td>69</td>
<td>0.0604</td>
</tr>
<tr>
<td>4.10.0–21-generic</td>
<td>4.10.0–24-generic</td>
<td>2097</td>
<td>1.8361</td>
</tr>
<tr>
<td>4.13.0–25-generic</td>
<td>4.13.0–21-generic</td>
<td>99</td>
<td>0.0882</td>
</tr>
<tr>
<td>4.10.0–28-generic</td>
<td>4.10.0–30-generic</td>
<td>32,607</td>
<td>28.545</td>
</tr>
<tr>
<td>4.10.0–28-generic</td>
<td>4.10.0–20-generic</td>
<td>20</td>
<td>0.0175</td>
</tr>
<tr>
<td>4.10.0–28-generic</td>
<td>4.10.0–27-generic</td>
<td>29,530</td>
<td>25.8514</td>
</tr>
<tr>
<td>4.10.0–28-generic</td>
<td>4.10.0–26-generic</td>
<td>2273</td>
<td>1.9898</td>
</tr>
<tr>
<td>4.10.0–30-generic</td>
<td>4.10.0–27-generic</td>
<td>29,285</td>
<td>25.6364</td>
</tr>
<tr>
<td>4.10.0–30-generic</td>
<td>4.10.0–24-generic</td>
<td>102</td>
<td>0.0893</td>
</tr>
<tr>
<td>4.10.0–30-generic</td>
<td>4.10.0–26-generic</td>
<td>2163</td>
<td>1.8935</td>
</tr>
<tr>
<td>4.13.0–16-generic</td>
<td>4.13.0–21-generic</td>
<td>60</td>
<td>0.0534</td>
</tr>
<tr>
<td>4.10.0–20-generic</td>
<td>4.10.0–22-generic</td>
<td>149</td>
<td>0.1306</td>
</tr>
<tr>
<td>4.10.0–27-generic</td>
<td>4.10.0–22-generic</td>
<td>105</td>
<td>0.0919</td>
</tr>
<tr>
<td>4.10.0–27-generic</td>
<td>4.10.0–14-lowlateness</td>
<td>43</td>
<td>0.0376</td>
</tr>
<tr>
<td>4.13.0–19-generic</td>
<td>4.13.0–17-generic</td>
<td>214</td>
<td>0.1904</td>
</tr>
<tr>
<td>4.11.0–14-generic</td>
<td>4.11.0–13-generic</td>
<td>106,550</td>
<td>93.855</td>
</tr>
<tr>
<td>4.10.0–22-generic</td>
<td>4.10.0–14-generic</td>
<td>19</td>
<td>0.0166</td>
</tr>
<tr>
<td>4.10.0–22-generic</td>
<td>4.10.0–24-generic</td>
<td>28,825</td>
<td>25.2419</td>
</tr>
<tr>
<td>4.10.0–22-generic</td>
<td>4.10.0–26-generic</td>
<td>1569</td>
<td>1.374</td>
</tr>
<tr>
<td>4.10.0–24-generic</td>
<td>4.10.0–26-generic</td>
<td>1866</td>
<td>1.634</td>
</tr>
<tr>
<td>4.4.0–22-generic</td>
<td>4.4.0–24-generic</td>
<td>235</td>
<td>0.2161</td>
</tr>
<tr>
<td>4.10.0–19-lowlateness</td>
<td>4.10.0–14-lowlateness</td>
<td>1127</td>
<td>0.9962</td>
</tr>
<tr>
<td>4.13.0–21-generic</td>
<td>4.13.0–17-generic</td>
<td>214</td>
<td>0.1904</td>
</tr>
</tbody>
</table>

![Execution Time of codeid-elf for various kernels.](image-url)
signature generation and detection, and can be incorporated into the security infrastructure.

Acknowledgment

This work was in part supported by the NSF grant # 1623276.

References


Table 4
Table with Comparison of Codeid-elf against previous works with Linux Kernels.

<table>
<thead>
<tr>
<th>Method</th>
<th>Similarity of Derived Signatures</th>
<th>Signatures Derived From Memory Dumps</th>
<th>ASLR Support</th>
<th>Version Detection Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS-Sommelier</td>
<td>Low</td>
<td>Yes</td>
<td>√</td>
<td>100%</td>
</tr>
<tr>
<td>k-BinID</td>
<td>High</td>
<td>No, derived at Run-Time using VMI</td>
<td>X</td>
<td>100%</td>
</tr>
<tr>
<td>Image-Based Kernel-Fingerprinting</td>
<td>High/No depending on Similarity in implementation</td>
<td>No</td>
<td>√</td>
<td>100%</td>
</tr>
<tr>
<td>codeid-elf</td>
<td>Low</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>