Guest-Based System Call Introspection with Extended Berkeley Packet Filter

by

Huzaifa Patel

A thesis proposal submitted to the School of Computer Science in partial fulfillment of the requirements for the degree of

Bachelor of Computer Science

Under the supervision of Dr. Anil Somayaji

Carleton University

Ottawa, Ontario

September, 2022

their kindness is masquerade.

yearning to occupy one with false pretenses.

it's used to sedate.

I promise you'll get this when the sky clears for you.

Abstract

Acknowledgments

I want to express my heartfelt gratitude to my supervisor, Dr. Anil Somayaji for providing me with the opportunity to work on a thesis during the final year of my undergraduate degree. Unlike previous variations of the Computer Science undergraduate degree requirements, completing a thesis is no longer a prerequisite. Therefore, I prostualte it is a great privlidge and honor to be given the opportunity to enroll into a thesis-based course during ones undergraduate studies.

I did not have prior experience in formal research when I first approached Dr. Somayaji. Despite this shortcoming, it did not stop him from investing his time and resources towards my academic growth. Without his feedback and ideas on my framework implementation and writing of this thesis, as well as his expertise in eBPF, Hypervisors, and Unix based Operating Systems, this thesis would not have been possible.

I would like to commend PhD student Manuel Andreas from The Technical University of Munich for introducing me to the concept of a Hypervisor. Without him, I would not have approached Dr. Somayaji with the intention of wanting to conduct research on them. His minor action of introducing me to hypervisors had the significant effect of inspiring me to write a thesis on the subject. I also want to thank him for his willingness to endlessly and tirelessly teach, discuss and help me understand the intricacies of hypervisors, the Linux kernel, and the C programming language.

I would also like to thank Carleton University's faculty of Computer Science for their efforts in imparting knowledge that has enthraled and inspired me to learn all that I can about Computer Science.

I would like to extend my appreciation to the various internet communities which have provided the world with invaluable compiled resources on hypervisors, Unix based operating systems, eBPF, the Linux kernel, the C programming language, and Latex, which has helped me tremendously in writing this thesis.

Finally, I would like to thank my immediate family for their encouragement and support towards my research interests and educational pursuits.

Contents

Nomenclature

VM	Virtual Machine
KVM	Kernel-based Virtual Machine
OS	Operating System
VMI	Virtual Machine Introspection
CPU	Central Processing Unit
AMD-V	Advanced Micro Devices Virtualization
VT-x	Intel Virtualization Extension
MSR	Model Specific Register
VMM	Virtual Machine Monitor, analogous to a hypervisor
EFER	Extended Feature Enable Register
eBPF	Extended Berkeley Packet Filter
VMI	Virtual Machine Introspection
API	Application Programming Interface

Introduction

Cloud computing is a modern method of delivering computing power, storage services, databases, networking, analytics, artificial intelligence, and software applications over the internet (the cloud). Organizations of every type, size, and industry are using the cloud for a wide variety of use cases, such as data backup, disaster recovery, email, virtual desktops, software development, testing, big data analytics, and web applications. For example, healthcare companies are using the cloud to store patient records in databases. Financial service companies are using the cloud to power real-time fraud detection and prevention. Finally, video game companies are using the cloud to deliver online game services to millions of players around the world.

The existance of cloud computing can be attributed to virtualization. Virtualization is a technology that makes it possible for multiple different operating systems (OSs) to run concurrently, and in an isolated environment on the same hardware. Virtualization makes use of a machines hardware to support the software that creates and manages virtual machines (VMs). A VM is a virtual environment that provides the functionality of a physical computer by using its own virtual central processing unit (CPU), memory, network interface, and storage. The software that creates and manages VMs is formally called a hypervisor or virtual machine monitor (VMM). The virtualization marketplace is comprised of four notable hypervisors: (1) VMWare, (2) Xen, (3) Kernel-based Virtual Machine (KVM), and (4) Hyper-V. The operating system running a hypervisor is called the host OS, while the VM that uses the hypervisors resources is called the guest OS.

While virtualization technology can be sourced back to the 1960s, it wasn't widely adopted until the early 2000s due to hardware limitations. The fundamental reason for introducing a hypervisor layer on a modern machine is that without one, only one operating system would be able to run at a given time. This constraint often led to wasted resources, as a single OS infrequently utilized a modern hardware's full capacity. The computing capacity of a modern CPU is so large, that under most workloads, it is difficult for a single OS to efficiently use all of its resources at a particular time. Hypervisors address this constraint by allowing all of a system's resources to be utilized by distributing them over several VMs. This allows users to switch between one machine, many operating systems, and multiple applications at their discretion.

Due to exposure to the Internet, VMs represent a first point-of-target for security attackers who want to gain access into the virtualization environment [3]. A virtual machine isolated from the Internet may behave consistently over time. However, a VM exposed to the Internet is a new system every day, as the rest of the Internet changes around it [2]. As such, the role of a VM is highly security critical and its priority should be to maintain confidentially, integrity, authorization, availability, and accountability throughout its existance. The successful exploitation of a VM can result in a complete breach of isolation between clients, resulting in the failure to meet one or more of the aforementioned goals. For example, the successful exploitation of a VM can result in the loss of availability of client services due to denial-of-service attacks, non-public information becoming accessible to unauthorized parties, data, software or hardware being altered by unauthorized parties, and the successful repudiation of a previous commitment or action by a principal. Because of this, effective methodologies for monitoring VMs is required.

In this thesis, we present Frail, a KVM hypervisor and Intel VT-x exclusive out-of-vm virtual machine introspection (VMI) framework that enhances the capabilities of existing related VMIs. In computing, VMI is a technique for monitoring the runtime state of a virtual machine [4]. Frail is a VMI for (1) tracing KVM guest system calls, (2) monitoring malicious anomalies, and (3) responding to those malicious anomalies from the hypervisr level. Our framework is implemented using a combination of existing software and our own software. Firstly, it utilizes Extended Berkeley Packet Filter (eBPF) to safely extract both KVM guest system calls and the corressponding process that requested the system call. Secondly, it uses pH's [2] implementation of sequences of system calls to detect malicious anomalies. Lastly, we utilize our own software to respond to the observed malicious anomalies by slowing down the guest process responsible for the malicious anomaly. To our knowledge, Frail is the second VMI-based system that supports all three system call mechanisms provided by the Intel x86 architecture, and has been proven to work for Linux 64-bit KVM guests.

1.1 Motivation

Current computer systems have no default general-purpose mechanism for detecting and responding to malicious anomalies within KVM VMs. As our computer systems grow increasingly complex, so too does it become more difficult to gauge precisely what they are doing at any given moment. Modern computers are often running hundreds, if not thousands of processes at any given time, the vast majority of which are running silently in the background. As a result, users often have a very limited notion of what exactly is happening on their systems, especially beyond that which they can actually see on their screens. An unfortunate corollary to this observation is that users also have no way of knowing whether their system may be misbehaving at a given moment. For this reason, we cannot rely on users to detect and respond to malicious anomalies. If users are not good candidates for adequately monitoring our VMs for malicious anomalies, VMs should be programmed to watch over themselves through the hypervisor level. Due to a VMs characteristic of being highly security critical, we have turned to VMI to provide the necessary tools to help trace, monitor and resond to malicious anomalies found within KVM VMs.

1.1.1 Why Design a New Framework?

Related work, (which we will discuss further in a future section) Nitro is over 10 years old and only works for linux kernel version 2.0.001.

1.1.2 Why Out-Of-VM Virtual Machine Introspection?

VMI software can either be placed in each VM that is active and needs monitoring (in-vm monitoring), or it can be placed on the hypervisor level (out-of-vm monitoring) outside of any VM. In this section, we discuss the motivations of why we use out-of-vm monitoring to trace KVM guest system calls, monitor malicious anomalies, and respond to those malicious anomalies.

Out-of-VM monitors offer four key advantages over traditional in-VM VMI's: (1) isolation, (2) inspection, (3) interposition, and (4) adaptability [8]. Isolation refers to the property that out-of-vm VMI's are less susceptible to tampering [8]. Unlike in-vm VMI's, out-of-vm VMI's run at a higher privlige level than guest OS'. For this reason, the out-of-vm VMI's are isolated from the guest OS, which makes them both tamper-resistant and inaccessible by the guest OS [7]. By isloating a VMI from a guest OS, it allows the VMI to be immune from attacks that originate in the guest, even if the VMI is actively monitoring the attacked guest [7]. The second key advantage, inspection, refers to the property that allows the VMI to examine the entire state of the guest OS while continuing to be isolated [8]. out-of-vm VMI's run one layer below all the guest OSs, and on the same layer of the hypervisor. For this reason, the VMI is capable of efficiently having a complete view of all guest OS states (CPU registers, memory, devices, and more) [7]. A VMI isolated from the VM also allows for constant and consistent view of the system state even if a VM is in a paused state. For example, if a VMI was not isolated from a VM, the VMI would stop executing when a VM goes into a pause state. In terms of our VMI frail, isolation allows us to efficiently, and consistently trace KVM guest system calls and their corressponding process that requested the system call efficiently and without having to rely on a possibly compromised VM for this information [8]. The third key advantage is due to interposition, which in our context, is the the ability to inject operations into a running VM based on certain conditions. Due to the interposition of the hypervisor and the VMI, the VMI is capable of modifying any of the states of the guest OS and interfering with every guest OS activity. In terms of our VMI frail, interposition makes it easy to respond to the observed malicious anomalies by slowing down the process responsible for the malicious anomaly [7]. Out-of-vm VMI's are very adaptable due to their existance in only the host OS. As a result, to deploy our VMI Frail to the hypervisor level, the guest OS does not have to be modified to accomodate for the VMI. Instead, only the hypervisor needs to be modified once.

Although in-VM systems have been very successful with systems like [insert in-vm vmi here], they are more susceptible to three types of threats: (1) privlidge escalation, (2) spoofing, and (3) tampering [8]. In-vm VMI's are not isolated as well as out-of-vm VMI's, because they are executed at the same privlidge level as the VM(s) that they are protecting [9]. For this reason, malicious software (malware), such as kernel rootkits that escalate privlidge can allow an attacker to access a VMI, or software that a VMI depends on, even if it not meant to be accessed. After successful privlige escalation of a VM, an attacker can tamper all the components of a VMI. For example, attackers can tamper the tracing software that collects system call information and/or the corressponding process that requested the system call. They can also tamper the sequences of system calls, which is the monitoring tool that enforces the VMI's security policy. Attackers can also tamper software that the VMI depends on like logs, proc files, or any other state information of interest to the VMI with false data (or even the code responsible for generating the data). In-VM VMI's usually have to trust the entire guest OS kernel, which tends to have a huge code base. However, out-of-VM often only needs to trust the underlying hypervisor, which has a smaller code base. For example, the KVM hypervisor has less than one twelfth the number of lines of code than the Linux kernel; this smaller attack surface leads to fewer vulnerabilities. For instance, since our VMI depends on hooking specific hypervisor functions, attackers can modify the relevant symbols within the symbol table to bypass VMI properties. Also, if the VMI is deployed using a kernel module in kernel space or a normal process in user space, attackers can simply remove or shut down the monitoring module or process. As long as an attack results in the VMI to continue its normal execution (e.g., no crashes), the VMI system can successfully generate a false pretense to mislead the VMI that a VM state is not malicious, when in fact it is.

1.1.3 Why eBPF?

1.1.4 Why System Calls?

1.1.5 Why Sequences of System Calls?

Instead of system call sequences, a neural network implementation is a modern approach to solving the problem of detecting malicious processes. Although system call sequences requires a less complex implementation than that of a neural network implementation, we believe complexity does not equate to better. Our motivation for using an implementation based on system call sequences is because we believe it is still effective in detecting malicious processes.

1.2 Problem

1.2.1 The Semantic Gap Problem

When designing computer monitoring systems, one goal has always been to have a complete view of the monitored target and at the same time stealthily protect the monitor itself. One way to achieve this is to use hypervisor-based, or more generally out of virtual machine (VM)-based, monitoring. There are, however, challenges that limit the use of this mechanism; the most significant of these is the semantic gap problem.

In order to leverage the full potential that VMI provides, identifying and isolating the relevant guest operating system (OS) state information becomes crucial. This process requires some semantic knowledge about the guest and is referred to as the semantic gap issue [3]. Bridging this semantic gap has been classified into three fundamental view generation patterns [12]. One of these patterns relies on knowledge of the hardware architecture to derive semantic information about the guest OS. Making use of the hardware architecture allows one to construct mechanisms that are resistant to evasion attempts through a method called hardware rooting [13]. This makes hardware-based information extraction particularly interesting for security approaches that are intended to detect malicious activity within a monitored VM.

1.3 Approaching the Problem

1.4 Contributions

1.5 Thesis Organization

In this thesis, we will examine the design and implementation of our VMI, explore its security implications on the KVM hypervisor and its guests, and explore its impact on system performance.

Related work

Background

3.1 Virtual Machine Introspection

VMI describes the method of monitoring and analyzing the state of a virtual machine from the hypervisor level [Nitro].

3.2 Hypervisor

There are two ways a hypervisor can virtualize a machine:

A hypervisor runs Guest OS instructions either directly on the host's CPU, or on the host OS. In both scenarios, the goal of a hypervisor is to provide a software-controlled layer that resembles the host hardware. Hypervisors can be classified into two types that are dependent on how they to runs Guest OS instructions.

(1) Type 1 (bare metal) hypervisors, which runs Guest OS instructions directly on the host's hardware in order to control the hardware and monitor the guest OS. Typical examples of such hypervisors include Xen, VMware ESX, and Microsoft Hyper-V.

(2) Type 2 (hosted) hypervisors, which run within a traditional OS. In other words, a hosted hypervisor adds a distinct software layer atop the host OS, and the guest OS becomes a third software layer above the hardware. Well-known examples of type 2 hypervisors include KVM, VMware Workstation, VirtualBox, and QEMU.

Although the preceding type 1 and type 2 hypervisor classification has been widely accepted, it is not clear it insufficiently differentiates among hypervisors of the same type (e.g., KVM vs. QEMU).

KVM is not a clear case as it could be categorized as either one. The KVM kernel module turns Linux kernel into a type 1 bare-metal hypervisor, while the overall system could be categorized to type 2 because the host OS is still fully functional and the other VM's are standard Linux processes from its perspective.

There-fore, based on how the virtualization gets designed (hardware vs. software) and the guest OS and its application code is executed, we can have another type of classification of hypervisors that will be used throughout this thesis:

(1) Native hypervisors that directly push the guest code to execute natively on the hardware using hardware virtualization.

(2) Emulation hypervisors that translate each guest instruction for an emulated execution using software virtualization.

Examples of native hypervisors include Xen, KVM, VMware ESX, and Microsoft HyperV, and emulation hypervisors include QEMU, Bochs, and the very early versions of VMware-Workstation and VirtualBox (note that recent VMware-Workstation and VirtualBox are able to execute the guest OS code natively). Since there is no binary code translation involved, native hypervisor runs much faster than emulation hypervisor.

In this thesis, we will be solely on the KVM VM.

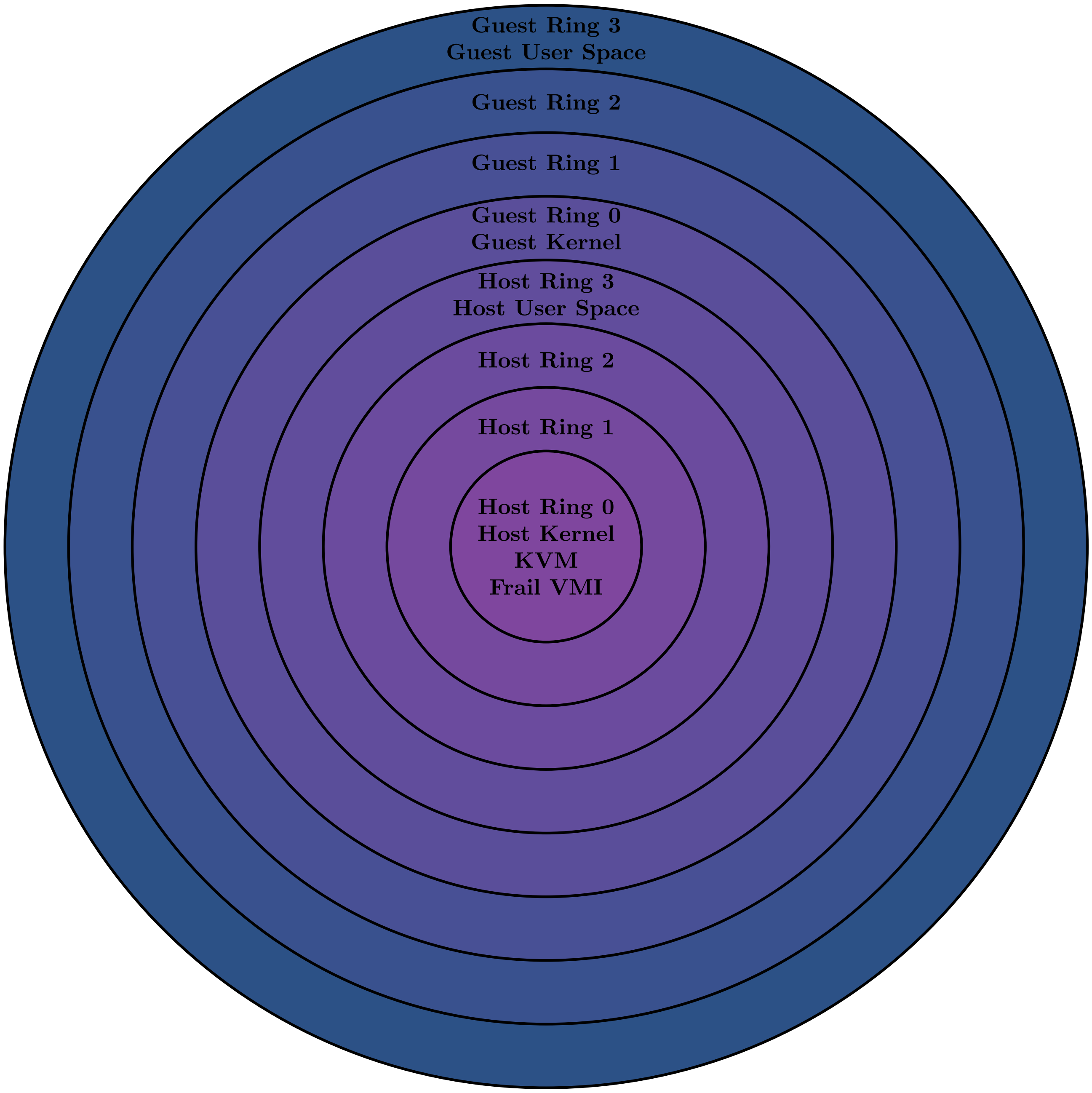
Hardware-assisted

3.3 Intel Virtualization Extention (VT-X)

Protection rings is a mechanism to protect data and functionality from faults (by improving fault tolerance) and malicious behaviour (by providing computer security). It is designed to have a hierarchical design that separates and limits the interaction between the user space and kernel space within and OS. It's purpose is to provide fault protection and tolerance among computer users, components, applications, and processes.

Ring 3 is the least privileged, and is where normal user processes execute. In this ring 3, you cannot execute privileged instructions. Ring 0 is the most privileged ring

Figure 3.1: Illustration of the x86 Protection Ring of a VM



that allows the execution of any instruction. In normal operation, the kernel runs in ring 0. Ring 1 and 2 are not used by any current operating system. However, hypervisors are free to use them as needed [6]. As shown in Fig. 1.1, the KVM hypervisor is kept in kernel mode (ring 0), the applications in user mode (ring 3), and the guest OS in a layer of intermediate privilege (ring 1). As a result, the kernel is privileged relative to the user processes and any attempt to access kernel memory from the guest Os program leads to an access violation. At the same time, the guest operating system's privileged instructions trap to the hypervisor. The hypervisor then performs the privileged instruction(s) on the guest OS' behalf.

3.4 The Kernel Virtual Machine Hypervisor

Kernel-based Virtual Machine (KVM) is a hypervisor that is implemented as a Linux kernel module that allows the kernel to function as a hypervisor. It was merged into the mainline Linux kernel in version 2.6.20, which was released on February 5, 2007. KVM requires a CPU with hardware virtualization extensions, such as Intel VT-x or AMD-V. While working with KVM, we will only be focusing on Intel VT-x hardware virtualization.

3.4.1 Model Specific Registers

3.4.2 VMCS

3.4.3 VM ENTRY Context Switch

3.4.4 VM EXIT Context Switch

3.5 QEMU

3.6 System Calls

3.7 Virtual Machine Introspection

In general, a security monitoring system can be defined as M(S, P) $\rightarrow$ True,False, (1) where M denotes the security enforcing mechanism, S denotes the current system state, and P denotes the predefined policy. If the current state S satisfies the security policy P, then it is in a secure state (True), and if M is an online mechanism, it can allow continued execution. Otherwise, it is insecure (False); an attack1 is detected, and M can halt the execution (for prevention) or report that there is an attack instance. For example, in an antivirus system, S can denote the current memory and disk state, and P the signatures of viruses; if M identifies that there is any running process or suspicious file having one of the signatures defined in P, the antivirus will raise an alarm. In a system call–based intrusion detection system, S can denote the current system call and P can denote the correct state machines for S; if M identifies that S deviates from P, then it can raise an intrusion alert.

3.8 eBPF

3.9 The Linux Kernel Tracepoint API

3.10 pH-based Sequences of System Call

There are 12 projects that use the guest-assisted approach. The pioneer work, LARES [Payne et al. 2008], inserts hooks in a guest VM and protects its guest com- ponent by using the hypervisor for memory isolation with the goal of supporting active monitoring. Unlike passive monitoring, active monitoring requires the interposition of kernel events. As a result, it requires the monitoring code to be executed inside the guest OS, which is why it essentially leads to the solution of inserting certain hooks inside the guest VM. The hooks are used to trigger events that can notify the hypervisor or redirect execution to an external VM. More specifically, LARES design involves three components: a guest component, a secure VM, and a hypervisor. The hypervisor helps to protect the guest VM component by memory isolation and acts as the communica- tion component between the guest VM and the secure VM. The secure VM is used to analyze the events and take actions necessary to prevent attacks.

3.11 Nitro: Hardware-Based System Call Tracing for Virtual Machines

Designing Frail

Implementing Frail

5.1 User Space Component

5.2 Kernel Space Component

5.2.1 Custom Linux Kernel Tracepoint

5.2.2 Kernel Module

5.3 Tracing Processess

5.4 Proof of Tracability of all KVM Guest System Calls

Threat Model of Frail

Future Work

Conclusion

References

https://dl.acm.org/doi/pdf/10.1145/361011.361073 [1] https://people.scs.carleton.ca/ soma/pubs/soma-
diss.pdf [2] https://dl.acm.org/doi/pdf/10.1145/2659651.2659710 [3] https://doi.org/10.1007/978-
1-4419-5906-5_647 [4] https://doi.org/10.1007/978-3-642-25141-2_7 [5] modern operat-
ing systems andrew s. tanenbaum [6] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=729
[7] https://dl.acm.org/doi/pdf/10.1145/1655148.1655150 [8] https://dl.acm.org/doi/10.1145/2775111
[9]