DRIVE: Dynamic Runtime Integrity Verification and Evaluation

Andre Rein
Huawei Technologies
German Research Center (GRC), Germany
andre.rein@huawei.com | ra@ra23.net

ABSTRACT

Classic security techniques use patterns (e.g., virus scanner) for detecting malicious software, compiler features (e.g., canaries, tainting) or hardware memory protection features (e.g., DEP) for protecting software. An alternative approach is the verification of software based on the comparison between the binary code loaded before runtime and the actual memory image during runtime. The expected memory image is predictable based on the ELF-file, the loading mechanism, and its allocated memory addresses. Using binary files as references for verifying the memory during execution allows for the definition of white-lists based on the actual software used. This enables a novel way of detecting sophisticated attacks to executed code, which is not considered by current approaches. This paper presents the background, design, implementation, and verification of a non-intrusive runtime memory verification concept, which is based on the comparison of binary executables and the actual memory image.

1. INTRODUCTION AND MOTIVATION

The complexity and number of cyber-attacks are increasing continuously over the last years. Especially during the last 5 years, very complex attacks were detected targeting a broad range of industries and governmental institutions (e.g. StuxNet (2010), Belgacom & Bengal Mobile (2013), Symantec (2014), German Bundestag (2015), Ruag AG (2016)). Those sophisticated attacks pose a substantial threat for many infrastructures due to their high complexity, evasiveness, and diversity, rendering effective detection exceptionally hard. Moreover, they are particularly tailored to circumvent broadly adopted and widely used counter-measures, such as firewalls, virus scanners and intrusion detection systems; thus, they are usually detected by coincidence after a long time past their initial deployment, e.g. by analyzing network traffic.

However, the behavior of a computer system, or a device acting as part of an IT-infrastructure, is defined by the software running on it. Thus, nearly every attack exploiting a software vulnerability interacts with the system memory to a certain extent. Often, devices are attacked by simply replacing or adding malicious software components permanently and execute them as desired. Those malicious modifications may occur both offline, for instance, by firmware manipulation, and online, usually by exploiting vulnerabilities during system runtime. However, the detection of persistent modifications on file system level is well researched; anti-virus/malware tools are available for more than 30 years and attestation of system states, based on integrity verification of loaded software, is also well understood.

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4944-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3052973.3052975

Still, the objective is not always to apply modifications on the file level; instead, system infiltration often utilizes runtime memory and control-flow manipulation to launch successful attacks. This means that the manipulation of executed software components is only volatile and without evidence permanently visible on the targeted device. However, even those non-permanent attacks leave trails of evidence, and thus can be detected. Memory Forensics (MF) enables the detection of maliciously tampered volatile memory. MF tools and techniques, capable of detecting even the most sophisticated attacks, can be used to analyze suspicious system behavior. Still, the field of MF is not thoroughly researched and well understood [25]. Nonetheless, certain MF tools exist [24, 33, 35] that facilitate a detailed systems' memory analysis. Nevertheless, using these tools require expert knowledge of MF, such as potential attack vectors, and are usually used to manually analyze a system after the detection of suspicious behaviors.

This paper presents Dynamic Runtime Integrity Verification and Evaluation (DRIVE). DRIVE enhances the classic onetime software component attestation approaches from load-time towards continuous and repeated monitoring and attestation throughout the entire software component's lifetime by adapting concepts from MF and System Integrity Verification. This enables secure software component runtime attestation and helps to detect advanced malicious threats. DRIVE also limits the adversary's capabilities by reducing the time span of detection of a successful and long lasting attack.

1.1 Goal and contributions

This work makes the following technical contributions:

- A non-intrusive and practical integrity measurement and verification concept to continuously monitor OS components' runtime memory content and meta-data
- A flexible architecture supporting different instantiations for secure measurement acquisition, measurement

anchoring in distinct Security Modules (SM) and integrity protected system state reports, comprising the system's current runtime state

- An significant enhancement to established static load-time measurement processes, enabling granular runtime measuring of memory artifacts
- A novel attestation scheme and techniques, providing integrity verification based on computable reference values for predictable runtime memory artifacts and meta-data analysis
- A complete prototype implementation of the presented measurement and verification concepts, evaluated on multiple different hardware architectures

1.2 Outline

Section 2 describes the adversaries intent, capabilities, and standard countermeasures of an expected system. After that, the system architecture is shown in Section 3, especially, the overall system, the individual architecture components, and details on instantiated measurement and verification architecture are described. Section 4 describes the technical details of the measurement, reporting and verification process. Thereafter, the implementation of the instantiated architecture is discussed in Section 5 providing metrics of the implementation and performance effects for the sensitive measurement process. In Section 6, related work is presented and Section 7 provides a conclusion and presents future research directions.

2. THREAT AND TRUST MODEL

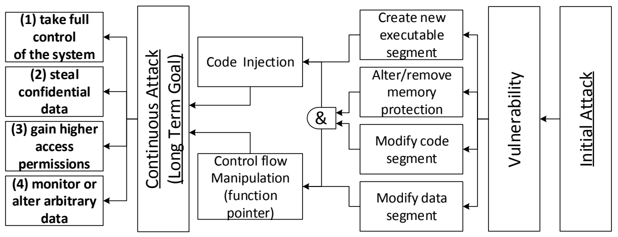
2.1 Threat Model

The adversary's initial goal is the injection and execution of arbitrary code in volatile memory for further exploitation. Specifically, as depicted in Figure 1, the adversary is assumed to launch an attack with particular long term goals such as (1) take control of a system (e.g. by launching a shell) or network; (2) steal confidential data (e.g. cryptographic keys or passwords); (3) gain higher access permissions (to circumvent access control mechanisms); or (4) monitor or alter arbitrary data (e.g. data of a program, network traffic, routing tables, etc.). Consequently, the target of interest is not the initial code injection but rather long term system modification and monitoring.

Moreover, it is assumed that the adversary wants to remain hidden in order to carry out malicious actions for as long as possible. Therefore, the attacks, such as code injection or control flow manipulation, are injected into the system memory to carry out their malicious behavior consistently. We assume the attacker may utilize the following different attack techniques: Create new executable segments, i.e., load new, map existing and execute shell-code in the virtual address space; alter or remove memory protection mechanisms, e.g. disable or circumvent Data Execution Prevention (DEP) ([1, 14]) mechanisms; modify code segments by changing or adding instructions or alter function pointers directly[1]; or modify data segment jump tables, i.e., altering memory jump addresses in the Global Offset Table (GOT,`.got`).

Furthermore, it is assumed that the adversary has access to a known exploit enabling a successful initial attack on the

[1]In order to launch a successful code segment manipulation attack, memory protection must be disabled/circumvented beforehand



(1) take full control of the system
(2) steal confidential data
(3) gain higher access permissions
(4) monitor or alter arbitrary data
Continuous Attack (Long Term Goal)
Code Injection
Control flow Manipulation (function pointer)
&
Create new executable segment
Alter/remove memory protection
Modify code segment
Modify data segment
Vulnerability
Initial Attack

Figure 1: Threat Model: Long Term Goals, Threats and Attack Techniques

system. This may be enabled by any vulnerability that allows memory access or code execution, e.g. buffer overflow, array over-indexing, or format string vulnerability. The adversary may, for instance, utilize a successful code reuse attack (e.g. ROP, JOP or SOP), or facilitate any other kind of attack in order to inject or load arbitrary data into volatile memory and exploit it arbitrary times [4, 27, 6]. Additionally, it is assumed that the adversary can successfully disable or circumvent other well-known defensive mechanisms such as Stack Canaries [36] or Address Space Layout Randomization (ASLR) [34]. As an example for a complex threat scenario, we assume an attack that overwrites memory addresses to function pointers in the `.got` of a user-process in order to permanently modify the control flow of an application, as described by Roglia [28].

2.2 Trust Model and Security Assumptions

DRIVE is a non-intrusive solution that aims to improve the overall system security; thus, previously mentioned security mechanisms are compatible to DRIVE and should be enabled as they provide significant obstacles for any adversary.

Nonetheless, DRIVE depends on some security requirements. Most importantly, DRIVE relies on security mechanisms that successfully detect or prevent execution of illicitly modified ELF-Files such as executables, libraries, kernel modules, and the kernel. In particular, DRIVE assumes that the system boots into a well known and reliable state. This requirement can be enforced by a secure or, at least, verified a measured boot of the OS as described in Trusted Computing [22]. As a result, the adversary must not be able to modify and load system binaries on disk, tamper or replace OS components, or disable those mechanisms without being detected. Although, DRIVE can be enhanced to detect those modifications as well, this work focuses solely on the measurement and verification of monitored components' runtime memory representations. Similar to well known integrity protection schemes, such as the Integrity Measurement Architecture (IMA) initially described by Sailer et. al [29], DRIVE facilitates a tamper-proof SM, for instance a Trusted Platform Module (TPM), in order to continuously record, track, and report a system state securely and arbitrarily. Most importantly, DRIVE relies on the tamper-resistance of the SM. That is, once measurements are anchored in the SM, they must not be changed.

In addition to that, one security-sensitive part of DRIVE relies on the measurement accumulation of the system memory before the measurement is anchored in the SM. For this reason, we assume that the attacker cannot interfere with the measurement process or disable it altogether. This means, the measurement component is considered immutable, i.e., isolated from the attacker. A straightforward implementation may employ OS based process isolation or user/kernel-level isolation; but, as soon as threats are considered that effectively bypass kernel level security protection mechanisms [21, 38, 9], more sophisticated isolation techniques and mechanisms must be used. For instance: (1) Virtualization- or hypervisor-based approaches; (2) Sandboxing approaches; (3) Hardware-backed approaches, like ARM TrustZone and Intel Software Guard Extensions (SGX) ; or (4) discrete hardware security co-processor based approaches.

DRIVE by itself is not limited to a particular isolation mechanism; the exact implementation mainly depends on use-case specific requirements and conditional architecture and platform capabilities. As a result, for this work we do not focus on the exact isolation mechanism and assume that kernel-level isolation is perfectly sufficient.

3. DRIVE HIGH LEVEL CONCEPT AND ARCHITECTURE

The main objective of DRIVE is to repeatedly measure, report, and verify runtime information present in system memory. In this section, the high level architecture of DRIVE is introduced, necessary components identified, and mechanisms for successful information acquisition, reporting and verification defined and discussed.

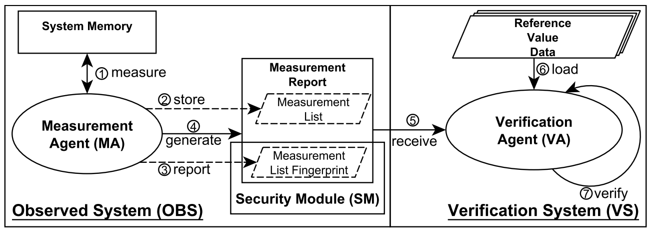


Figure 2: DRIVE Architecture for Measurement, Reporting and Verification.

The high level architecture, as depicted in Figure 2, shares some similarities with the architecture described in [29]. DRIVE utilizes similar conceptual measurement and verification agents, and also a SM. The measurements and the reported data methods utilize cryptographic hash functions; however, measurement acquisition and verification differs significantly, because DRIVE repeatedly measures and verifies the runtime system's states including dynamic information, whereas Sailers' work only considers static one-time measurements taken before the loading process.

3.1 System Overview

The architecture consists of two individual systems: The Observed System (OBS) and the Verification System (VS). OBS and VS can be implemented either in the same or, preferably, in different logically or physically isolated contexts. The OBS implements a Measurement Agent (MA) – responsible to acquire the desired measurements from the memory – that produces a Measurement Report, comprising of a list of related measurement, anchored in a Security Module (SM).

The VS implements the Verification Agent (VA), which receives a Measurement Report for verification. The VA implements the actual verification process, under the assistance of Reference Value Data. The verification process is called Remote Attestation (RA) in accordance with TCG terminology. Accordingly, the VS can be operated by a Trusted Third Party (TTP) which means the verification process can be base on certain assumptions, i.e. the VS and all its reference data are considered as reliable.

3.2 Architecture Components

Measurement Agent: Measurement and Reporting of Measurements. The MA is the core component of the OBS and responsible for secure measurement and reporting. DRIVE's Measurement and Reporting architecture is depicted in Figure 3 and will be presented in the following.

In order to conduct a secure measurement process, DRIVE's Measurement Component implements following operations: ① Measure and receive targeted memory contents; ② append the individual measurements to an ordered Dynamic Measurement List (DML); and, ③ generate fingerprints, representing and protecting the DMLs integrity and anchor the fingerprints in the SM.

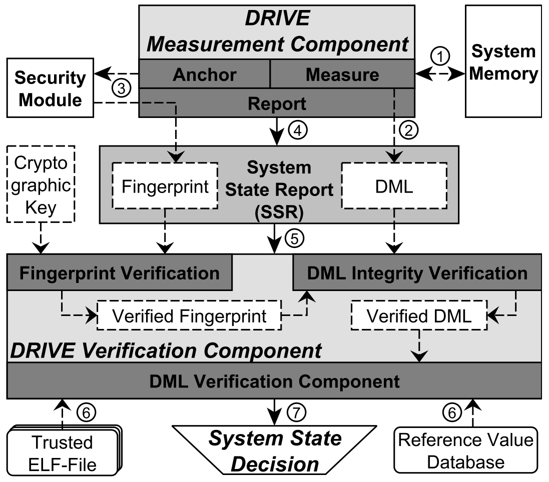


Figure 3: DRIVE Measurement and Reporting Process

In addition to the secure measurement process, the MA also implements one additional operation: ④ generation of a System State Report (SSR) for the reporting process. The operation itself generates the SSR by adding the anchored fingerprint from the SM and the maintained DML. The SSR is used by the VA during the verification process that will be described next.

Verification Agent: Verification of Measurements. The VA implements all necessary functionality to evidently verify the measurements, securely embedded in the previously described SSR. Involved operations include receiving a SSR from an OBS, loading of necessary Reference Value Data and verifying the SSR based on these information.

The SSR is received by the VA either by direct introspection of the OBS or, in preferred cases, received during Remote Attestation, which is done by establishing a secure data channel between OBS and VS. That means the secure channel must provide integrity, authenticity, confidentiality and freshness. Once the SSR was successfully received ⑤, SSR data is verified against the SM anchored integrity value [cf. Figure 3]. First of all, the authenticity of the security anchored fingerprint is verified by the Fingerprint Verification. This involves: Loading a well known or signed cryptographic key and verifying the authenticity of the fingerprint. The DML Integrity Verification component calculates a cryptographic hash sum based on the DML and compares this self calculated fingerprint against the previously Verified Fingerprint. Only if both values are equal, the integrity of the received DML is guaranteed; accordingly, the DML is considered untampered and is therefore regarded as a Verified DML. A more detailed description of this particular process for the RV calculation is provided in Section 4.5.

In order to determine whether the measurements were modified during runtime, the VA compares every measured value present in the DML against references. Every successfully verified measurement draws the conclusion that the measured memory part did not change unexpectedly and thus can be considered as trustworthy. The process of DML verification is depicted in Figure 3 and briefly described next.

The verification of the DML' measurements is as follows: ⑥ For every individual measurement in the DML the Verification Component tries to generate or find valid Reference Values either by simulating the load process of Trusted ELF-Files or searching a valid precomputed Reference Value. In case a valid reference value was determined, the individual measurement represents a well-known state. Subsequently, the process is repeated for every other measurement in the DML. If all measurements were verified successfully, the measurements are considered as reliable, accounted by ⑦ the System State Decision. In addition to the aforementioned steps, further verification steps, based on different meta-data, such as access permissions, sizes, etc., can also be applied during the verification. The verification process is further described in Section 4.6.

4. MEASUREMENT AND VERIFICATION CONCEPT

This section introduces the technical measurement and verification concepts of DRIVE. In particular, we will present a technical solution to continuously measure, report and verify measurements, acquired during the runtime of different software components in both user- and kernel-space. Due to its design, DRIVE is not affected by concurrency, ASLR, stack protection or dynamic library pre- or late-loading mechanisms, all these concepts are fully supported and considered. Still, concrete implementations of the solution may utilize different methods and components for measurement acquisition and algorithms during verification; however, the general concepts remain similar for different hardware and system architectures.

4.1 Behavior and Predictability of Object Code Fragments

DRIVE's concept relies on two inevitable requirements: (R1) the ability to measure ELF' segments or sections in memory and (R2) verify the measurements with a reduced set of available information based on white-lists of reliable information.

In many cases, R1 is a weak requirement that mainly depends on the correct access permissions determined by the OS configuration. This means, by fulfilling the preconditions for being able to read targeted memory areas, it is possible to acquire any desired measurement. Still, one major issue in this regard is to identify certain designated memory areas, which will be discussed later in Section 4.3.

The major problem DRIVE solves is to fulfill requirement R2, i.e. the successful verification of acquired measurements. Many approaches [13, 35, 24, 3] solve this problem by taking a measurement and simply compare this measurement to a previously acquired runtime reference measurement that was taken in an assumed or known reliable system state. In contrast to this, DRIVE does not rely on such initial reference measurements on the targeted system, but solely on current runtime information and involved well-known ELF files. This enables the definition of a reliable system state externally and totally isolated and independent from the targeted system. For this reason, DRIVE depends on the predictability of measured memory areas, whereas the predictability depends on the runtime behavior. Table 1 depicts individual encapsulated ELF sections according to their dynamic behavior. The categories are `(s)tatic` and `(d)ynamic` and the predictability is classified by `(p)redictable` or `(u)npredictable`.

Table 1: Access Permissions and Runtime Behavior of Runtime ELF Segments and Encapsulated Sections

VAS Segment		ELF Section		
Name	**Perm**	**Name**	**Perm**	**Type**
`.text`	`r-x`	`.text`	`r-x`	`sp`
		`.init`	`r-x`	`sp`
		`.plt`	`r-x`	`sp`
		`.rodata`	`r--`	`sp`
`.data`	`rw-`	`.data`	`rw-`	`du`
		`.bss`	`rw-`	`du`
		`.got`	`rw-`	`dp`
`Heap`	`rw-`	`--`	`--`	`du`
`Stack`	`rw-`	`--`	`--`	`du`

Permissions: (r)ead, (w)rite, e(x)ecute

Type: (d)ynamic, (s)tatic, (p)redictable (u)npredictable

As expected, there exists a correlation between non-writable sections and static runtime behavior. This means, a section or segment which is not writeable, is not expected to change during its runtime[2]. Therefore, as a general rule, if a segment is not writable, its behavior is static and predictable. Consequently, the entire `.text` segment is both static and predictable and fulfills R2. This means a successful verification is self-evident. Verification of `.text` segments will further be discussed in Section 4.5.

In contrast, the `.data` segment is flagged writeable and encapsulates both predictable and unpredictable sections. Therefore, the segments' content behaves dynamic and changes during runtime arbitrarily. Still, not all encapsulated sections are considered unpredictable. While the `.data` and `.bss` sections are, the `.got` section, a table involved in function address resolution, is considered as predictable and thus can also be verified by DRIVE. The `.got` section verification processes will be discussed further in Section 4.5. As an example of dynamic and unpredictable segments, the `heap` and `stack` segments are worth to be mentioned. Due to their fast changing and arbitrary behavior, their contents cannot be verified by DRIVE.

As explained, the behavior and predictability of whole segments or the individual sections affects the measurement and the verification processes. While static segments can be measured as a single instance, dynamic sections must be measured individually. However, verification processes of dynamic sections are only applicable if the contents are predictable. As a result, the verification of unpredictable dy-

[2]There exist hot-patching mechanisms and self modifying code which actually change read-only segments legitimately during runtime. Both concepts are left to be analyzed and researched in future work.

namic sections or segments is considered impracticable for DRIVE and thus not considered.

In addition to the behavior, Table 1 also shows the access permissions of the mentioned segments and lists their designated ELF object code sections encapsulated inside the segment, along with their individual access permissions, i.e. `(r)ead`, `(w)rite` and `e(x)ecute`. In specific circumstances, mappings with `rwx` permissions exist and are indeed necessary as, e.g., Virtual Machine and interpreter based programming languages often require access permissions considered as insecure.

DRIVE facilitates access permissions as an indicator for potential threats and therefore measures them as meta-data. The meta-data is also analyzed during the verification phase and if unexpected changes to access permissions are detected the system is considered as compromised and becomes untrusted. This is especially important, but not limited, to unpredictable segments where content verification cannot be applied. Meta-data manipulation can cause severe harm to the system security [31] and, for this reason, is considered very important and treated seriously by our approach. Meta-data verification is further discussed in Section 5.1 and 4.5.

4.2 Relocatable and Position Independent Code

Generally, object code can be categorized into two different variants influencing the loading process of ELF and the transformation into a ready-to-run state. There exists: (1) Relocatable Code (RCC), which depends on fixed memory addresses that, requires transformation by the linker or loader prior to their execution; on contrary, (2) Position Independent Code (PIC) does not rely on fixed memory addresses and, thus, can be executed from arbitrary memory addresses. In this section we will discuss both variants briefly and explain how both variants affect the DRIVE concept.

Relocatable Code. RCC is the standard mechanism for executable object code in user-space, the Kernel, and LKM in kernel-space. Necessary relocations that rely on fixed memory addresses are resolved either during the object code linking phase or during the load-time under assistance of the dynamic linker/loader [17]. More precisely, executable object code – or in other words applications – in user-space and the OS kernel are link time relocated; they use and rely on fixed loading addresses and therefore can already be relocated during the linking phase [18]. For the link time relocated object code, the instructions already contain the concrete target addresses for all symbols. This means the `.text` segment object code within the ELF is equal to the object code in the memory segment. LKMs, on the other hand, are loaded at arbitrary dynamic addresses. For this reason, they must be relocated during their loading process. The LKM ELF contains a specific symbol table and the object code uses placeholders for all referenced symbols. During the loading process, the loader analyses the symbol table, resolves the target addresses of the symbols and patches the object code placeholders with resolved addresses[3]. In case of LKM loading, symbols from both the Kernel and other LKMs must be resolved and are considered.

Although RCC plays a negligible role for shared libraries today, the process of symbol resolution and object code patching is similar as described for LKMs. Instead of the LKM loader, `ld` is responsible for relocation, whereas symbols from other libraries are the main target during the symbol resolution process.

As a result, the verification of load-time relocated RCC depends on dynamic addressing similar to the `.got` verification, discussed in Section 5.1. Thus, in order to verify load-time relocated RCCs, the symbol resolution and patching process must be simulated during verification, considering loading addresses of required external dependencies. This means, the technical verification solution extracts the `.text` segment from the ELF, resolves symbol addresses and patches them into the related placeholders. More technical details about this process are described in Section 4.5.

Position Independent Code. Generating PIC for shared libraries is the default behavior in any modern Linux distribution. All shared libraries in Linux are indeed PIC.

During the loading process of PIC, the `.text` segment remains unchanged. Therefore, the whole `.text` segment remains identical to its counterpart in the ELF object code. Regarding DRIVE, this means that the measurement and verification of the `.text` segment does not depend on loading memory addresses and thus can be derived directly from the object code segments of the ELF. The major benefit of PIC is the possibility to share `.text` segments between multiple processes. As long as the `.text`-segment of a library does not changeit is shared to any process in the system, and thus resides only one time in physical memory. Considering that many libraries are shared between all system processes, PIC saves a considerable amount of physical memory.

Recently, so called Position Independent Executables (PIE) are also getting more and more attention and are also seen in many recent Linux distributions. Both PIC and PIE hugely benefit from ASLR, because they do not rely on fixed addresses and therefore render possible exploitations harder, because concrete memory addresses are often required to conduct certain attacks. In addition to that, PIC code also facilitates the dynamic loading of shared libraries during runtime, which is used by many applications with the assistance of the `dlopen` system call. From the perspective of DRIVE, both PIC and PIE are equal, thus, the measurement and verification of corresponding `.text` segments is fully supported.

Global Offset Table. Both, PIC and link time RCC in user-space is usually paired with a mechanism called lazy-binding [17], implementing an on-demand function symbol resolution and relocation process. Technically, address resolution involves the maintenance[4] of a table, i.e. the Global Offset Table (`.got`), to store memory addresses of targeted external symbols. As shown in Table 1, the `.got` is considered as dynamic, but predictable. Therefore, DRIVE measures the `.got` section that is located inside the `.data` segment and also provides a verification mechanism that generates a process specific `.got` by simulating the address resolution process. The `.got` symbol resolution mechanism is similar to LKM RCC symbol resolution and rather complex; therefore, we provide a concrete example in Section 5.1.

4.3 Measurement of Memory Mapped Segments and Sections

Conducting a measurement of entire segments in memory is simple given just the sufficient access permissions. Every

[3]The patching process is also often called fix-up.

[4]Managed by the Procedure Linkage Table `.plt` as part of the `.text` segment.

segment resides in a certain memory area that is identified by a well-known address. Accordingly, once the targeted memory address and the size of the to-be-measured memory area is determined, the measurement process executes designated functions provided by the OS. In contrast to established static integrity measurement concepts, such as IMA, DRIVE is not limited to onetime measurement and thus can conduct its measurements anytime an repeatedly after the software component is fully loaded. This means the measurement process can be hooked to or triggered after relevant system calls, such as `mprotect` or `dlopen`, or run on a timer, executing the measurements on a defined time interval.

However, measuring individual sections, more precisely the `.got` section inside the `.data` segment, is more complex – especially for shared libraries– due to ASLR. To actually measure the `.got`, its start address and size must be identified. However, both are maintained in internal data-structures that also depend on assigned runtime addresses. For this reason, start address and size must be determined during the measurement process itself. This means, in order to measure the `.got`, the relevant in memory ELF section, i.e. the `.dynamic` section, must be located, analyzed and interpreted correctly. Based on this information the allocated memory area of the `.got` can be calculated and finally measured.

DRIVEs' measurement process facilitates cryptographic hash functions (CHF) to provide secure and reliable representations of the measured contents. The concept of applying CHF to measurements is well-known; however, no known related work applies CHF to measure and verify individual memory segments or sections. In fact, CHF based measurement and verification of runtime memory contents with dynamic behavior are not considered at all in prior or recent work. Still, the information that is measured can be designed in a flexible way, tailored to the particular memory area. In principle, meta-data, such as the memory start address (msa), size (ms), access permissions (map) and, if available, related filename or module name (mf) is gathered and added to all measurements for every measured segment. In addition to this meta-data, for every predictable memory area, i.e. currently the `.text` segments and `.got` sections, a hash digest (mhd) of the measured content is created and added to the measurement.

All measurements are stored in an ordered Dynamic Measurement List (DML) whereas every single list entry forms a set S with variable information, for instance:

$$S = \{mf, msa, ms, map, mhd\}$$

As a result, a Measurement Set MS entails individual sets of S for every measured memory area:

$$MS = \{S_0, S_1, \ldots, S_n\}$$

Thus, the DML is composed of one or multiple MS:

$$DML = \{MS_0, \ldots, MS_n\}$$

As explained, the DML and its comprised data is later used as part of the verification scheme. A concrete instantiation of the measurement concept is presented in Section 5.

4.4 Reporting of Measured Data

In the following, the mechanisms and construction of the anchored and reported data is briefly described.

In order to report measured data between MA and VA, we introduce a Runtime System State Report (RSSR) comprising the DML along with a SM anchored fingerprint.

Therefore, the measurement process implements an additional reporting mechanism to anchor the measured sets in the SM. That is, for every individual set in MS a hash digest is calculated and concatenated with its predecessor to form a Security Module Anchored Fingerprint $SMAF$, such that[5]:

$$SMAF_n = digest(digest(MS_{n-1})|MS_n)$$

As a result, $SMAF_n$ reflects a fingerprint that comprises all measurement sets MS in a single value. Thus, $SMAF_n$ provides integrity protection over every individual measurement. Moreover, we define $RSSR$ that simply merges the current DML and its fingerprint $SMAF$, such that:

$$RSSR = \{SMAF_{DML} \cup DML\}$$

As explained, $RSSR$ gets securely transmitted from OBS to VS in order to be used during the verification mechanism. The described process for reporting and the transmission is part of a Remote Attestation Protocol and uses similar building blocks for its construction.

4.5 Verification of Reported Measurement Data

The DRIVE verification concept is based on the idea of unaltered reference measurements. As previously described, the measurement process calculates a hash over predictable measured memory areas. This means, in case the verification process can (1) calculate an expected hash digest (ehd) based on a reliable source and (2) successfully compare ehd against the measured hash digest mhd, DRIVE can determine whether mhd was illicitly altered or not. In the following section, a brief description of the individual verification steps is provided.

DML Integrity Verification. The first step in every verification process of DRIVE is the integrity verification of the given DML only using the information from RSSR. Since every intermediate measured hash digest was extended to the SM, in which we ultimately establish trust in, we can use this value to verify DML's integrity.

Therefore, DRIVE calculates the expected $ESMAF$ based on the reported DML comprised Measurement Sets $MS_{0\ldots n}$ and verifies whether the calculated value is equal to the reported $SMAF$.

$$ESMAF_n = digest(digest(MS_{n-1})|MS_n)$$

If, and only if, the expected value is equal to the reported, i.e. $ESMAF_n == SMAF_n$, the DML is considered as reliable and was not tampered. This implies that the DML represents the currently measured runtime system state at the time the report was generated. After the integrity of the DML is assured, the verification process continues to verify every individual measurement.

4.6 DML Content Verification

The verification of the DML encompasses all individual measurements $MS_{0\ldots n}$ depending on the content of the measured object code. In this section, verification schemes for predictable segments or sections are explained. The verification is based on the principle of RV calculation derived from ELF files using the (1) extraction of designated meta-

[5]"|" denotes concatenation

Table 2: Program header Excerpt from */bin/bash*

Type	Offset	Size (Dec)	Flags
LOAD	`0x00000000`	978908 Bytes	`R-E`
LOAD	`0x000efdf0`	36536 Bytes	`RW-`

data, segments and sections from ELF files, (2) composition and modification of extracted content to represent measured memory dependents, and (3) calculation and comparison of RVs against reported information to make a decision about the reliability of the OBS.

ELF Object Code Extraction. All information necessary to extract the targeted section is present in the `ELF-Header`. In Table 2, an excerpt from the Program Header of the ELF executable */bin/bash* on a X86_64 system is shown. The type "LOAD" instructs the loader to load the object code into memory. The relevant information DRIVE uses are (1) the "Offset", determining the position of the first to-be-loaded object code byte and (2) "FileSize" determining the number of to-be-loaded bytes relative to the given position.

In order to calculate the reference value for the relevant segment, the object code is extracted as described and saved to a temporary object code file $tocf$. $tocf$ is then used as a basis for the following reference value calculation.

Reference Value Generation for PIC and link-time RCC. The reference value generation for PIC (RV_{PIC}) and link-time RCC consists of calculating the hash digest of $tocf$: $RV_{PIC} = digest(tocf)$. While RV for any PIC and link-time RCC does not depend on dynamic information, it stays constant for every process instantiation. Thus, calculated RV are persistently stored on the VS, for instance in a database and can be extracted and calculated independently from the verification process.

Reference Value Generation for Global Offset Tables. The verification of `.got` relies on different information about the measured process. Specifically, all processes' shared library code sections' memory addresses must be known during verification. The other necessary information is contained in the ELF-files involved in the process execution and gets extracted during verification.

The verification itself depends on the re-calculation of the GOT based on the library memory start addresses (msa_{lib}), the resolved symbol's offset ($offset_{symbol}$) and the GOT's location address $GOT_{address}$[6].

The required memory loading addresses msa_{lib_n} are part of the SSR's DML, as depicted in Figure 4, and available during verification. The GOT's location address $GOT_{address}$ and the symbol name can be derived directly from the ELF-file's relocation section's header (`.rela.plt`). Most importantly, the symbol file offset (sfo) can be derived from the symbol tables, i.e. `.symtab`, of referenced library ELFs by their symbol name. Once the required information is extracted, the expected `.got` entry address can be calculated by:

$$GOT_{asa} = msa_{lib} + sfo$$

The process is repeated for every single relocation entry in the order given by $GOT_{address}$. Once all got addresses are calculated and sorted, CHF can be applied in order to calculate the `.got` reference value RV_{GOT} as follows:

$$RV_{GOT} = digest(GOT_{asa_0} || GOT_{asa_{\dots}} || GOT_{asa_n})$$

[6]which is used to determine the order of the `.got` table.

Finally, it can be compared against the measured hash digest of the `.got` (mhd_{GOT}).

Reference Value Generation for RCC. To calculate reference values for RCC, a correlation between the loaded ELF-File and the reported information MS must be established. The process itself is quite similar to the described `.got` verification. The first step during the ad-hoc reference value calculation is the extraction of the related memory start address msa from MS. After msa is extracted and $tocf$ loaded by the verification program, the loading process of the object code is simulated using the following steps: (1) Load relocation rel from ELF relocation sections, i.e. `.rela.text` or `.rela.dyn`, (2) extraction of the symbol file offset sfo from rel, (3) location of the referenced symbol offset rso in ELFs symbol tables, i.e. `.symtab`, (4) calculation of the absolute symbol address $asa = msa + rso$, and (5) patching $tocf$ at position sfo with asa.

Once every relocation is applied in $tocf$, the hash digest can be calculated: $RV_{RCC} = digest(tocf)$. RV_{RCC} is only valid for the given RCC with the concrete msa. Regarding LKM, where this process is usually applied, this means that unloading and reloading the LKM would render the RV_{RCC} outdated, because the start address would most likely change. In this case, RV_{RCC} must again be recalculated on basis of the new msa.

Reference Value Generation for Meta-data. Meta-data information such as expected file-sizes and access permissions are also extracted from the ELF. As shown in Table 2, "size" and "flag" information is also available in the program header. Therefore, this information is extracted and persistently stored as meta-data.

Verification of individual DML entries. Usually, the verification of individual DML entries is the core verification process that either uses persistent reference values RV_{PIC} or triggers ad-hoc reference value calculation to derive RV_{RCC}. The verification process iterates over the whole DML and compares the measured object code to the RVs for every individual entry. If, and only if, the complete DML can be verified successfully, the OBS is considered as reliable (or authentic).

5. IMPLEMENTATION AND EVALUATION

DRIVE's current measurement implementation measures the OBS' runtime-state for both kernel- and user-space. In order to demonstrate the applicability of our conceptual work, this section describes all utilized components and gives details about their implementation. Furthermore, measurement and verification components are evaluated during a simulated attack on a well known user-space application. Finally, performance critical operations are identified and challenges, discovered during the evaluation, presented.

5.1 Runtime Integrity Measurement Component

DRIVE's runtime integrity measurement component (RIMC) is implemented as a LKM and measures kernel and user-space without any access restrictions. Please note, we chose a LKM to demonstrate the feasibility of the concept and to provide measurement results for user and kernel-space on multiple architectures. As mentioned in Section

3.2 and 2.2, a production grade implementation must consider additional security measures to run at higher privilege levels or isolated, by utilizing virtualization-, sandbox- or hardware-based MA implementations, if available. During the measurement process, RIMC utilizes the kernel internal data-structures for process management and MM. RIMC currently supports (1) the measurement of the OS kernel and all loaded LKMs, in kernel-space and (2) the measurement of all active processes, including executable and shared object code. RIMC is not architecture specific and was evaluated on the X86_64, PPC32 and ARM64 platform.

The user-space process measurement in RIMC iterates over all running processes utilizing the internal kernel structures `mm_struct` and `vm_area_struct` to identify active processes and determine their mapped segment memory addresses. As mentioned, the identification of the `.got` section is more complex. Here, RIMC first locates the `.dynamic` section from the program headers and, afterwards, identifies the location and size of the `.got` section based on the information found at the `.dynamic` section. After that, RIMC measures the individual process memory areas as described in Section 4. Finally, RIMC appends the measurements to the DML and anchors the fingerprint to a TPM by calling the `TPM_extend()` functionality. The kernel-space DML is generated similar to the algorithm for a single process. Still, the data-structures for LKM are organized in a list utilizing the `module` struct and the Kernel solely relies on `mm_struct`[7].

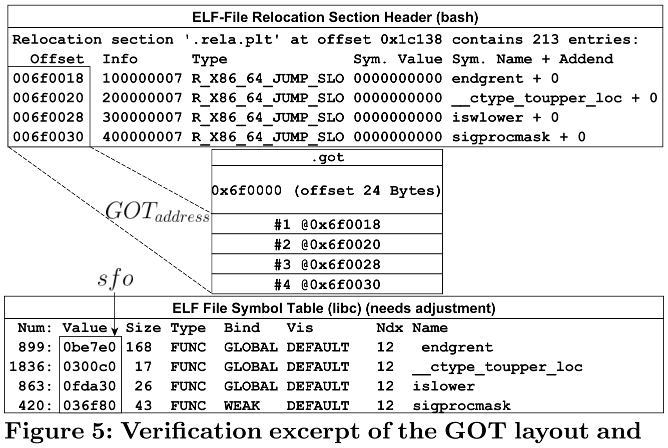
```
---------------------------------------------------------
Userspace DML for /bin/bash process:
name              address (msa)  size perm hash digest
/bin/bash         000000400000   956k r-x  1c5b27f74..
/bin/bash.got     0000006f0000   1722 rw-  bc09bbbb8..
libc-2.13.so      7fbd32f4b000  1772k r-x  79e937e84..
libc-2.13.so.got  7fbd33309000    114 rw-  48d78c75a..
---------------------------------------------------------
```

Figure 4: Excerpt of DML for a `/bin/bash` user-space process on the X86_64 platform

Figure 4 depicts an excerpt for the accumulated user-space DML for the `/bin/bash` process. Once the measurements are successfully appended to the designated DML and reported to the TPM, they can be verified by a VS.

Runtime Integrity Verification Component. As previously described in Section 4.5, the verification basically consists of two parts. The first verifies the integrity of the received DML and the other verifies the individual accumulated measurements inside the DML. The DML integrity verification mechanisms is implemented similar to the well researched Stored Measurement Log (SML) integrity verification schemes (cf. [29, 26]), therefore we will not further discuss this mechanism. Instead, this work focuses on the second part, i.e., integrity verification of individual DML measurements.

During the evaluation, we developed a tool gathering RV and meta-data from ELF-files in firmware images or by given paths. The tool implements ELF object code extraction mechanisms and RV generation as previously described. Furthermore, the tool collects additional meta-data from ELF such as expected access permissions and sizes; moreover, the tools determines whether an ELF-file is PIC or link-/load-time-relocated RCC. After the information is successfully gathered, the tool stores all verification data persistently



ELF-File Relocation Section Header (bash)

```
Relocation section '.rela.plt' at offset 0x1c138 contains 213 entries:
  Offset  Info      Type              Sym. Value Sym. Name + Addend
006f0018  100000007 R_X86_64_JUMP_SLO 0000000000 endgrent + 0
006f0020  200000007 R_X86_64_JUMP_SLO 0000000000 __ctype_toupper_loc + 0
006f0028  300000007 R_X86_64_JUMP_SLO 0000000000 iswlower + 0
006f0030  400000007 R_X86_64_JUMP_SLO 0000000000 sigprocmask + 0
```

```
.got
0x6f0000 (offset 24 Bytes)
#1 @0x6f0018
#2 @0x6f0020
#3 @0x6f0028
#4 @0x6f0030
```

ELF File Symbol Table (libc) (needs adjustment)

```
 Num: Value  Size Type  Bind   Vis      Ndx Name
 899: 0be7e0 168  FUNC  GLOBAL DEFAULT   12  endgrent
1836: 0300c0  17  FUNC  GLOBAL DEFAULT   12 __ctype_toupper_loc
 863: 0fda30  26  FUNC  GLOBAL DEFAULT   12 islower
 420: 036f80  43  FUNC  WEAK   DEFAULT   12 sigprocmask
```

Figure 5: Verification excerpt of the GOT layout and symbol resolution for `/bin/bash` 4.3 application.

in a $SQLite$[8] database. In addition to that, the tool also maintains copies of load-time-relocated RCC ELF-files on the file-system for RV calculation. Once the tool finishes the data collection, the RVs are ready to be used during the verification process to either find valid RVs for PIC and link-time-relocated RCC or provide ELF-files for the ad-hoc RV and `.got` calculation process.

Furthermore, we analyzed three different distributions[9] for six different architectures[10] in order to determine the distribution of PIC and relocated RCC code. As it turned out, all shared libraries exclusively utilize PIC. Additionally, we confirmed most executable ELF-files still use link-time RCC relocation; yet, we identified Position Independent Executables (PIE) in all analyzed systems [cf. Table AT3]. Regarding kernel-space, load-time-relocated RCC still plays a major role for LKM. All LKMs analyzed used load-time RCC and are thus relocated during initial loading.The Kernel images are, as expected, all statically linked.

PIC and link-time-relocated RCC Verification. The verification process of PIC and link-time-relocated RCC follows the concept described in Section 4.5. The technical realization extracts mhd from S and searches for a mhd RV. As a result, the measured object code is considered trustworthy if and only if a RV is found that is equal to mhd; consequently it was not illicitly modified by an adversary. This simple approach covers everything necessary to verify the object code integrity of PIC and link-time-relocated RCC object code. As expected, the integrity verification can also be combined with additional meta-data verification, described below.

Global Offset Table Verification. The verification of the `.got` is an important operation and always applied for PIC and link-time-relocated RCC by DRIVE. As depicted in Figure 5, the symbol name and the GOT's location address $GOT_{address}$ can be extracted from the ELF-file of the GOT target.

For instance, the `endgrent` function used by `/bin/bash` is implemented in `libc-2.19.so` and its designated GOT address $GOT_{address}$ is `0x006f0018`. By analyzing the `libc-2.19.so` ELF-file, the `endgrent` symbol resolves to the offset `0x0be7e0`, relative to the loading address `0x7fbd32f4b000`

[7]All struct definitions can be found in *Linux Source*/include/linux/mm_types.h

[8]https://www.sqlite.org/

[9]Ubuntu, Debian and Fedora

[10]X86, X86_64, ARM32, ARM64, PPC32 and PPC64

[c.f. Figure 4, 5]. Thus, the resolved address for the `endgrent` function is `0x7fbd330097e0`.

Based on this, Table 3 shows the calculated GOT which is in line with the measured GOT. It has to be noted that before the CHF can be applied on the calculated GOT, the target architecture's endianess must be considered; this means, if the target architecture uses little-endian, the resolved addresses must be converted into that format.

Load-time-relocated RCC Verification. We implemented the RCC load-time relocation only for LKMs, because they are the only components that utilize this functionality nowadays. During verification, at first, the LKM ELF-file is loaded from the persistent ELF storage. Once loaded, the described mechanisms are applied: extract the `.text` segment from the ELF-file and save it in a temporary file; obtain the relocation, analyze it, and calculate the symbol target address based on DML information; finally, patch the calculated symbol address into the temporary file. Afterwards, calculate the expected hash digest *ehd*, and compare it to *mhd*. The measured object code is considered trustworthy if and only if both values are equal.

It has to be noted that LKM utilize additional indirection through trampoline-jump tables in certain cases [11]. Those mechanisms are very architecture specific and hence out of scope for this work. Nevertheless, we successfully implemented DRIVE's concept as described in Section 4.5, corresponding to the architecture specific behavior.

Meta-data Verification. Meta-data verification is applied for every measurement and considered as very important for non-reproducible dynamic segments and sections where content based measurements cannot be applied, i.e., in particular `data.data`, `data.bss`, `stack`, and `heap`, but also conclusive for all other memory mapped segments. As soon as an adversary is able to modify memory access permissions, every memory mapping can potentially be used for successful exploitation. For instance, executing shell-code requires access permissions (`rwx`) for `stack`, `heap` or anonymous mappings, which is generally forbidden and thus detected by DRIVE's meta-data verification.

Kernel Image Verification. As previously mentioned, the Linux kernel image is statically linked to predefined addresses. This means its `.text` segment does not depend on addresses determined during runtime; thus, it is possible to pre-calculate a RV for the kernel, add its RV and compare *mhd* from the DML to the RV. Accordingly, we implemented a mechanism calculating a kernel RV hash digest and our verification component detects illicit runtime tampering of the kernel's `.text` segment. But, in contrast to statically linked user-space objects, the kernel implements sophisticated fix-up mechanisms, self-applied very early during its initial loading phase, as described by Kittel et al. [16]. These fix-ups are architecture dependent and specific to the particular CPU and MMU of the system. Therefore, we re-implemented necessary mechanisms to mimic these fix-ups in order to pre-calculate reference values for our evaluated systems . As it turned out, once the fix-ups are applied, the kernel's RV remains constant for a particular hardware configuration and, thus, can be reused. It has to be noted that we disabled mechanisms that facilitate runtime code patching (i.e. jump labels and ftrace). However, we are aware of these mechanisms and will consider them in future work.

[11]E.g. in PPC 32 certain relative jumps exceed a maximal jump-length of 24 Bit for a target symbol address. This can only occur in kernel-space, due to its size.

5.2 Concept and Implementation Evaluation

In this section, we evaluate the implementation for the DRIVE concept and verify that different attack techniques are detected by their designated verification modules.

According to our threat analysis presented in Section 2, we evaluated the `/bin/bash` application regarding the mentioned attack techniques, as follows:

(1) *Create new executable segment*: Load an unreferenced and unknown[12] library in the process memory utilizing `dlopen` system-call.
(2) *Alter/remove memory protection mechanisms*: Utilize the `mprotect` system-call to change the permissions of the `stack` and `heap` from `rw-` to `rwx` and `.text` segment from `r-x` to `rwx`.
(3) *Modify code segment*: Change permissions of `.text` segment to `rw-`, replace instruction for the `endgrent()` function call with a `nop` sledge, reset permissions of `.text` segment to `r-x`.
(4) *Modify data segment jump tables*: Replace pointer of the `endgrent()` lib-c function with `setgrent()`[13] in the `.got .data` segment.

It has to be noted that we did not exploit a vulnerability of `/bin/bash`. In order to keep the experiment simple, we went for an attack-simulation approach instead. However, we assume that a sophisticated attacker can easily utilize any mentioned technique, given an initial needed vulnerability[14]. For this reason, we developed a small python tool, utilizing the `ptrace()` system-call for attaching to a process[15]. The tool was used to simulate the attacks and apply the modification directly in the process memory of an instantiated `/bin/bash` user-process. After the attack was simulated, a measurement and verification process was conducted.

As depicted in Table 4, the different attack techniques are detected from different or even multiple verification modules. This means, for the verification of an user-space process, all aforementioned verification steps must be executed in order to detect all defined attack techniques. First of all, the meta-data of each measured segment is analyzed and compared against stored meta-data RVs. In particular, the size and permission flags are verified for each measurement. The meta-data verification module was able to detect attack (1) and (2) but was not able to detect attack (3) and (4). The next step is the verification of the `.text` segment's integrity, based on a pre-calculated RV. The verification is successful if the a RV_{PIC} is found that is equal to the measured value *mhd*. Pre-calculated RV verification detected (1) and (3) but not (2) and (4). Finally, as explained, every verification process also involves the verification of the `.got` related to the designated `.text` segment. The expected RV_{GOT} is calculated and compared against the measured value *mhd*. If both values are equal, the verification is considered successful. Ad-hoc `.got` relocation detected attack (4) but not (1), (2) and (3).

[12]unknown means in this case that no RV was calculated for that library

[13]`endgrent` and `setgrent()` use `void` as their parameter

[14]We honestly hope there is no exploitable vulnerability in the analyzed bash version.

[15]In addition to that, the `/proc/sys/kernel/yama/ptrace_scope` was set to `0` enabling process hooking by `ptrace`

Table 3: GOT calculation for `/bin/bash` application

symbol name	sfo	library name	msa_{lib}	$GOT_{address}$	GOT_{asa}
`endgrent`	0x0be7e0	`libc-2.19.so`	0x7fbd32f4b000	0x006f0018	0x7fbd330097e0
`__ctype_toupper_loc`	0x0300c0	`libc-2.19.so`	0x7fbd32f4b000	0x006f0020	0x7fbd32f7b0c0
`iswlower`	0x0fda30	`libc-2.19.so`	0x7fbd32f4b000	0x006f0028	0x7fbd33048a30
`sigprocmask`	0x036f80	`libc-2.19.so`	0x7fbd32f4b000	0x006f0030	0x7fbd32f81f80

Table 4: Results for the evaluation of the different attack techniques for the `/bin/bash` user-process

Verification Module	(1)	(2)	(3)	(4)
Meta-data Verification	−	−	+	+
Pre-calculated RV Verification	−	+	−	+
Ad-hoc .got relocation RV Verification	+	+	+	−

−: integrity verification failed (modification detected)
+: integrity verification successful (no modification detected)

In this section, we evaluated the functionality of the individual verification modules. As shown, all verification modules behave as expected, and detected the attacks that were intended to be detectable. Similar to the described user-space process verification, the verification modules for the Kernel and LKM, i.e. load-time-relocated RCC and kernel image verification module, were also evaluated. We used a writeable `/dev/mem` device to simulate the modification of the kernel image's and different LKMs' `.text`[16] segments. Afterwards, we performed a measurement and verification that detected the modifications in both cases as expected.

5.3 Security Analysis

As demonstrated in the previous Section, DRIVE is able to detect attacks on different kinds of levels and with different granularity. Code Injection/Manipulation Attacks, like, for instance, (1) *Create new executable segment* and (3) *Modify code segment* in predictable memory areas, are very well suited for DRIVE's employed detection mechanism. As a result, DRIVE can be used to verify whether a predictable memory region was modified or not. Similarly, Code Pointer Modifications in designated and predictable memory areas, such as (4) *Modify data segment jump tables*, can also be detected reliably. Further identification of possible predictable areas, for instance well-known predictable kernel data-structures such as the (2) `system call table` are left for future research. The meta-data verification used to detect *Alter/remove memory protection mechanisms* also has the potential to detect sophisticated attacks to the system. Especially in the field of unpredictable memory areas, meta-data analysis could provide enough evidence to make a decision about the system state. However, self-contained Code Reuse Attacks, such as ROP and its different variants are currently not detectable by DRIVE, because they usually do not modify predictable memory areas. Control Flow Integrity (CFI) and Code Pointer Integrity (CPI) mechanisms are tailored to detect those kind of attacks. Hence, it would be very interesting to study and analyze those protection mechanisms and possibly integrate some concepts into DRIVE or vice versa. Very recent research on attacks that solely utilize non-control data to implant malicious actions seem to be resistant even against CFI and CPI. However, if data-structures are altered that rely on static information or modify meta-data that can be measured and verified successfully, DRIVE can possibly be used to detect at least specific variants of those attacks.

[16]Similar to attack technique (3)

5.4 Measurement and Verification Micro-benchmark

This section presents a micro-benchmark for the measurement and verification process of the described implementation. We identified two time-critical operations utilizing the Kernels *ftrace*[17] debugging mechanism: (1) The hash calculation of the individually measured memory segments and (2) TPMs `extend()` operation. Afterwards we added measurement code to the implementation to get precise results, it turned out both (1) hash calculation and (2) tpm `extend()` consume roughly 98% of the overall time. The metrics for none[18], sha-1 and sha-256 hash algorithms were accumulated on two different architectures and platforms, i.e. X86-64 Bit Intel Core i5-4570 CPU @ 3.20GHz on a standard Ubuntu 14.04 (3.13 Kernel) server installation (X86_64) and 32 Bit PPC e500mc @ 1.2 GHz (4 cores) on Windriver embedded Linux (2.6.34 Kernel) (PPC32).

Based on the implementation the measurement is executed without interrupting and, thus, not directly affecting the performance of the measured processes, thus, concrete values were not quantifiable. Obviously, the measurement affects the overall system performance negatively; however, the performance impact is considered to be negligible on the experimental setup. A test under heavy system load would have distorted the benchmark randomly. Thus, in order to get reproducible, comparable and meaningful micro-benchmark results, we decided against experiments under heavy load.

As depicted in Table AT1, the metrics show the distribution of the average computational time for (1) code and library measurements and (2) TPM `extend()` operation for one completed measurement cycle. For X86_64, $\sim$ 28 different code and $\sim$ 393 library segments ($\sim 96 - 97$ MB), aggregated from $\sim$ 70 individual libraries were measured. Similarly, for PPC32 $\sim$ 30 different code and $\sim$ 698 library segments ($\sim 277 - 282$ MB), aggregated from $\sim$ 230 individual libraries. Considering that libraries' unmodified `.text` segments are mapped one-time in physical memory, multiple measurements of the same library is unnecessary. This leaves a huge optimization potential by implementing a more sophisticated measurement strategy that considers deduplication of already conducted measurements.

As previously described in Section 4.3, the TPM `extend()` operation is only applied one-time for every individual process and consumed $\sim 10.9 - 11.1$ ms on X86_64 and $\sim 15.19 - 15.3$ ms on PPC32 on average, independent from utilized hash algorithms. In addition to the overall time consumption, we also analyzed page access times, the overall measured size, and total time consumption on average. As expected, the relation between measured size to consumed

[17]https://www.kernel.org/doc/Documentation/trace/ftrace.txt
[18]https://git.kernel.org/cgit/linux/kernel/git/herbert/cryptodev-2.6.git/tree/crypto/crypto_null.c

time behaves almost linear for all hash functions on both architectures.

Most interesting is the overall computational time used for the hash calculation, because these represent the predominant part that the CPU is occupied during the measurement process. For X86_64, none $\sim 0.0291s$, sha-1 $\sim 0.3947s$ and sha-256 $\sim 0.6315s$; for PPC32, none $\sim 1.0010s$, sha-1 $\sim 7.6695s$ and sha-256 $\sim 7.7194s$ respectively. The TPM `extend()` operation, anchoring the measurements in the TPM, takes $\sim 11ms$ on X86_64 and $\sim 15.25ms$ on PPC32. Please note that this does not affect the computational effort; the TPM executes this operation without the assistance of the CPU.

With regards to the verification process, prior research by Rein et. al analyzed integrity verification of a SML [26]. The verification of the DML integrity via TPM Quote is identical to our described approach. Effects from the larger individual data-sets for our measurements during template verification are considered to be negligible. As explained in Section 5.1, for PIC measurements the verification relies on a simple comparison of the measured hash digest mhd and the reference value RV. Therefore, verification times are expected to be equal to the referenced SML verification and depend mainly on the implementation details, whether parallelized or not, and the used database back-end. In contrast to that, DML `.got` and RCC verification involves ad-hoc reference generation for the measured `.got` and LKMs. Following the description presented in 5.1, we identified the symbol resolution process to be the most time consuming operation in our implementation; It took $\sim 1.53s$ to generate the symbol table for the `.got` of the `/bin/bash` application and $\sim 1.66s$ to generate the initial symbol table for the LKMs. Once the symbol tables were generated, the remaining operations, i.e. calculation of correct jump addresses, `.got` generation (`/bin/bash` application), the patching process (LKM), and hash calculation, took for the `/bin/bash` application $\sim 14ms$ and for a single LKM $\sim 52ms$.

In this section we presented some metrics regarding the performance of the PoC implementation for both, the in-memory measurement process and the verification. Most notably, we observed in all cases a linear dependency between computational effort and the actual size of the taken measurement or the amount of values to be verified.

6. RELATED WORK

MF and extraction tools are available for different OS. For instance, LIME [33] targets the extraction of memory images, whereas the Volatility Framework [35] and FATKit [24] additionally support further analysis of the extracted memory contents. Memory extraction is usually implemented as a LKM using the internal kernel APIs and data-structures (c.f. 5.1). Volatility is considered the most recent and advanced tool and provides a huge amount of analysis plug-ins [19]. Usually, MF is used to manually analyze the behavior of infected systems if there is initial suspicion of misbehavior. In other words, it is currently not used to automatically detect or report malicious behavior; therefore, SMs are not considered in either of the aforementioned solutions.

Since the initial design and proposal of the TCG-based Integrity Measurement Architecture (IMA) [29] from Sailer et. al, certain other approaches about static and dynamic integrity measurement were researched and published. IBMs IMA implementation is widely used today and considered the standard procedure for static measurement and attestation. However, the initial concept does not consider measurements taken during runtime.

Subsequent work utilized different techniques to gather and verify measurements on the targeted system parts. Especially Linux kernel rootkit detection was a particular target of research interest. Hardware based solutions were described in [13, 3] demonstrating the utilization of a hardware based co-processor in form of a PCI extension card. Both works take snapshots of the memory and verify the snapshots by an external system. LKIM [20, 23] was published as another approach for Kernel Integrity Monitoring; LKIM utilizes similar mechanisms to our presented work considering reference value generation, which they called base-lining; i.e., generating cryptographic hash values based on simulation of the loading process of kernel and LKMs. Regarding dynamic LKM behavior, LKIM does not detail its base-lining and verification mechanisms. All the same, LKIM does not facilitate a SM for secure anchoring and reporting of measurements; in fact, no details about the verification process were published.

In addition, many hypervisor-based approaches were researched [30, 37, 10, 8, 2, 32, 15]. Most recently, measurement of a virtual machine was demonstrated in [5], utilizing Linux paging mechanisms by hypervisor introspection. Both, measurement and verification, are performed by the hypervisor. The measurement acquisition is similar to our work, however, the proposed solution employs intrusive enforcement mechanisms relying on anterior verification. The solution utilizes a TPM for verification, nonetheless, concrete verification mechanism were not published. Moreover, the role and exact tasks of the TPM were not explained.

Furthermore, hardware-backed solutions were also used for the similar introspection mechanisms. Specifically, SPROBES [11] utilized ARM TrustZones to enforce invariants detecting malicious modifications to Linux kernel code during runtime. The invariants are used to evaluate different meta-data to deduce whether the system was maliciously altered or not. Consequently, the invariants are chosen and designed such that unintended modification may not occur undetected.

7. CONCLUSION AND FUTURE WORK

In this work we presented DRIVE – a novel approach complementing MF and System Integrity Verification concepts in order to produce evidence of a system state and attest the reliability of a measured system's runtime configuration. The described approach enables continuous monitoring of long running OS components, and thus reduces the attack surface for sophisticated adversaries by detecting attacks aiming the volatile memory. Our presented architecture is flexible and can be adapted in varied ways. We demonstrated an exemplary instantiation of our proposed architecture and provided a security evaluation and metrics that demonstrate DRIVE's applicability.

Currently, we are in the process of enhancing the concept and technical details of DRIVE. Our current research targets the following major challenges: (1) Implementation and integration of DRIVE's MA into an isolated environment for secure measurement acquisition. At present, an ARM TrustZone based implementation and a LKM for extended hypervisor introspection is developed and evaluated; (2) Support and integration of kernels hot-patching and runtime code patching mechanisms; (3) Measurement and verification of kernel critical data-structures targeted in DKOM based attacks; (4) Mechanisms to trigger measurement on security critical events; (5) Performance improvements reducing the computational effort and time, especially during the measurement acquisition process.

8. REFERENCES

[1] S. Andersen and V. Abella. *Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies.* 2004.

[2] A. M. Azab et al. "HIMA: A hypervisor-based integrity measurement agent". In: *Computer Security Applications Conference, 2009. ACSAC'09. Annual.* IEEE. 2009, pp. 461–470.

[3] A. Baliga, V. Ganapathy, and L. Iftode. "Detecting kernel-level rootkits using data structure invariants". In: *Dependable and Secure Computing, IEEE Transactions on* 8.5 (2011), pp. 670–684.

[4] T. Bletsch et al. "Jump-oriented Programming: A New Class of Code-reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ASIACCS '11. Hong Kong, China: ACM, 2011, pp. 30–40. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966919. URL: http://doi.acm.org/10.1145/1966913.1966919.

[5] C. Chang et al. "Research on Dynamic Integrity Measurement Model Based on Memory Paging Mechanism". In: *Discrete Dynamics in Nature and Society* 2014 (2014).

[6] S. Checkoway et al. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM conference on Computer and communications security.* ACM. 2010, pp. 559–572.

[7] T. Committee et al. "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2". In: *TIS Committee* (1995).

[8] J. Criswell et al. "Secure virtual architecture: A safe execution environment for commodity operating systems". In: *ACM SIGOPS Operating Systems Review.* Vol. 41. 6. ACM. 2007, pp. 351–366.

[9] CVEdetails.com. *Linux Kernel Vulnerabilities.* https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[10] T. Garfinkel et al. "Terra: A virtual machine-based platform for trusted computing". In: *ACM SIGOPS Operating Systems Review.* Vol. 37. 5. ACM. 2003, pp. 193–206.

[11] X. Ge, H. Vijayakumar, and T. Jaeger. "Sprobes: Enforcing kernel code integrity on the trustzone architecture". In: *arXiv preprint arXiv:1410.7747* (2014).

[12] M. Gorman. *Understanding the Linux virtual memory manager.* Prentice Hall Upper Saddle River, 2004.

[13] N. L. P. Jr et al. "Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor." In: *USENIX Security Symposium.* San Diego, USA. 2004, pp. 179–194.

[14] V. Katoch. *Whitepaper on Bypassing ASLR/DEP.* http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf.

[15] C. H. Kim et al. "CAFE: A Virtualization-Based Approach to Protecting Sensitive Cloud Application Logic Confidentiality". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.* ASIA CCS '15. Singapore, Republic of Singapore: ACM, 2015, pp. 651–656. ISBN: 978-1-4503-3245-3. DOI: 10.1145/2714576.2714594. URL: http://doi.acm.org/10.1145/2714576.2714594.

[16] T. Kittel et al. "Code validation for modern os kernels". In: *Workshop on Malware Memory Forensics (MMF).* 2014.

[17] *ld(1) Linux User's Manual.*

[18] J. R. Levine. *Linkers and Loaders.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1558604960.

[19] M. H. Ligh et al. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory.* John Wiley & Sons, 2014.

[20] P. A. Loscocco et al. "Linux Kernel Integrity Measurement Using Contextual Inspection". In: *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing.* STC '07. Alexandria, Virginia, USA: ACM, 2007, pp. 21–29. ISBN: 978-1-59593-888-6. DOI: 10.1145/1314354.1314362. URL: http://doi.acm.org/10.1145/1314354.1314362.

[21] C. S. Ltd. *Whitepaper on Quadrooter.* URL: https://www.checkpoint.com/resources/quadrooter-vulnerability-enterprise/.

[22] C. Mitchell. *Trusted computing.* Springer, 2005.

[23] J. A. Pendergrass and K. N. McGill. "LKIM: The Linux Kernel Integrity Measurer". In: *Johns Hopkins APL technical digest* 32.2 (2013), p. 509.

[24] N. L. Petroni et al. "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory". In: *Digital Investigation* 3.4 (2006), pp. 197–210.

[25] A. Prakash et al. "On the Trustworthiness of Memory Analysis—An Empirical Study from the Perspective of Binary Execution". In: *Dependable and Secure Computing, IEEE Transactions on* 12.5 (2015), pp. 557–570.

[26] A. Rein et al. "Trust Establishment in Cooperating Cyber-Physical Systems". In: *Cybersecurity of Industrial Control Systems, Security of Cyber Physical Systems.* 2015.

[27] R. Roemer et al. "Return-oriented programming: Systems, languages, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), p. 2.

[28] G. Roglia et al. "Surgically Returning to Randomized lib(c)". In: *Computer Security Applications Conference, 2009. ACSAC '09. Annual.* 2009, pp. 60–69. DOI: 10.1109/ACSAC.2009.16.

[29] R. Sailer et al. "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *USENIX Security Symposium.* Vol. 13. 2004, pp. 16–16.

[30] A. Seshadri et al. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles.* SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 335–350. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294294. URL: http://doi.acm.org/10.1145/1294261.1294294.

[31] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines in ELF: A Spotlight on the Underappreciated Metadata". In: *Presented as part of the 7th USENIX Workshop on Offensive Technologies.* Washington, D.C.: USENIX, 2013. URL: https://www.usenix.org/conference/woot13/workshop-program/presentation/Shapiro.

[32] U. Steinberg and B. Kauer. "NOVA: a microhypervisor-based secure virtualization architecture". In: *Proceedings of the 5th European conference on Computer systems.* ACM. 2010, pp. 209–222.

[33] J. Sylve. "Lime-linux memory extractor". In: *Proceedings of the 7th ShmooCon conference.* 2012.

[34] P. Team. *PaX address space layout randomization (ASLR).* http://pax.grsecurity.net/docs/aslr.txt. 2003.

[35] Volatility_Foundation. *Volatility Framework.* URL: http://www.volatilityfoundation.org.

[36] P. Wagle, C. Cowan, et al. "Stackguard: Simple stack smash protection for gcc". In: *Proceedings of the GCC Developers Summit.* Citeseer. 2003, pp. 243–255.

[37] Z. Wang and X. Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity". In: *Security and Privacy (SP), 2010 IEEE Symposium on.* IEEE. 2010, pp. 380–395.

[38] W. Xu et al. "From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security.* CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 414–425. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813637. URL: http://doi.acm.org/10.1145/2810103.2813637.

APPENDIX

A. MICRO-BENCHMARK EXPERIMENTS

Table AT1 shows the detailed results we discussed in Section 5.4. The metrics consists of multiple measurements that were taken from the implemented DRIVE LKM conduction a measurement process.

B. TECHNICAL BACKGROUND

This section provides an overview of related technical principles used in DRIVE. DRIVE measures system memory content and meta-data. For this reason, it is deeply related to core low level functionalities provided by the OS kernel's Memory Management (MM). Hence, basic MM concepts are briefly introduced. In particular, we describe how the loading process utilizes the MM and how this affects the memory contents and meta-data. Furthermore, access permissions of Virtual Address Space (VAS) segments will be shown, basic memory sharing concepts described, and static and dynamic behavior of individual VAS segment sections discussed.

B.1 Loading Principle in Modern Operating Systems

Every component, meant to be executed on a computer system, follows a certain well-known approach. Leaving aside the concrete software components, every CPU based computation relies on data existent in the systems main memory. Even so, every different architecture comes with different concrete implementations for the process, all modern architectures follow the same principle and are based on similar organization of the involved data.

B.1.1 Organization of Object Code

In Unix based systems, the Executable and Linkable Format (ELF) standard defines the organizational structures of executables, shared libraries, and core dumps, referenced as object code during this work. Inside the ELF, different components of the object code are represented by different sections. Most notable sections are (1) the `.text`, `.init`, and `.plt` sections, encapsulating executable instructions, (2) the `.data`, `.rodata` and `.bss` sections, holding initialized and uninitialized data, and (3) `.got`, a table organizing data structures to related function symbols. Additionally, there exist more sections in the ELF, for example, representing program and section headers, procedures, and symbols resolution tables. A comprehensive overview is given in [7]. Figure AF1 shows a user-space process ELF to VAS mapping example. Please note, once the section is loaded and resides inside memory in VAS, we will use the term segment to identify ELF content loaded into memory. As shown, multiple related sections from the ELF are organized in a single segment. This relation is specified in the section-to-segment mapping in the ELF header and can be different for every ELF.

B.1.2 Object Code Loading

Before object code instructions can be executed by the CPU all necessary object code must reside in memory. Depending on the object code, different loaders are responsible for the loading process. Following the boot process, we briefly describe the loading process of ELF beginning with the systems' Bootloader. The result of the loading mechanism is depicted in Figure AF1.

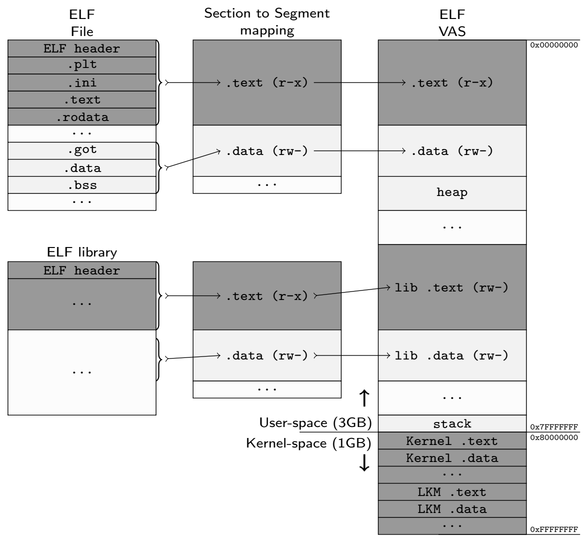


Figure AF1: ELF-file to VAS mapping with aid of Section to Segment Mappings

Bootloader and Kernel Setup. The Bootloader instructs the CPU to load the OS Kernel object code into memory at a fixed location. From this point onwards, the Kernel takes complete control over the MM. If present, in a very first step the Kernel initializes the Memory Management Unit (MMU) and sets up internal structures to organize the mappings between the physical and virtual memory (VM) that is the Page Table [12]. Most importantly, the VM is separated into Kernelspace and Userspace in this process. Afterwards, the Kernel continues with its execution until a Loadable Kernel Module (LKM) needs to be loaded.

Kernel Module Loader. The Kernel Module Loader (KML) loads the requested LKM ELF object code into memory and transforms the LKM into a ready-to-run state. During this process, dependencies between different LKMs are resolved. More precisely, the KML (1) inspects every unresolved symbol, either referencing procedures or data from the LKM itself, the Kernel or other LKMs; (2) resolves the symbol by determining the targeted Virtual Memory Address (VMA) and (3) patches the determined target VMA directly into the in-memory object code instructions (relocation), present in LKMs' `.text` segment. After all executed transformations, the LKM is in a ready-to-run state and can be used as intended. It has to be noted, that the relocation processes are architecture dependant. They rely on the architecture specific relocation types, defined in the architecture's specifications found at http://refspecs.linuxbase.org/elf/. Still, only certain relocation types are relevant for the KML. For instance, for X86_64[19]: `R_X86_64_64`, `R_X86_64_32`, `R_X86_64_32S`, `R_X86_64_PC32` and PPC32[20]: `R_PPC_ADDR32`, `R_PPC_ADDR16_LO`, `R_PPC_ADDR16_HI`, `R_PPC_ADDR16_HA`, `R_PPC_REL24`. For ARM32, ARM64 and PPC64 there are too many relevant relocation types to list them.

[19]c.f. http://lxr.free-electrons.com/source/arch/x86/kernel/module.c#L173

[20]c.f. http://lxr.free-electrons.com/source/arch/powerpc/kernel/module_32.c#L237

Table AT1: Metrics for Measurement Component

System	(1) Code/Library Segments				(2) TPM Extend Function			(1 + 2) Cumulative		
hash/arch	time	percent	pagetime	size	time	percent	single	time	percent	overall
none X86_64	$0.0291s$	8.21%	$1.71\mu s$	$97.06MB$	$0.3177s$	89.47%	$10.96ms$	$0.3468s$	97.68%	$0.3551s$
sha1 X86_64	$0.3947s$	54.44%	$17.30\mu s$	$96.41MB$	$0.3188s$	43.98%	$11.07ms$	$0.7135s$	98.42%	$0.7249s$
sha2 X86_64	$0.6315s$	66.25%	$27.77\mu s$	$97.06MB$	$0.3169s$	33.25%	$10.93ms$	$0.9484s$	99.50%	$0.9532s$
none PPC32	$1.0010s$	64.77%	$14.90\mu s$	$279.33MB$	$0.4591s$	29.71%	$15.30ms$	$1.4601s$	94.48%	$1.5454s$
sha1 PPC32	$7.6695s$	92.65%	$109.75\mu s$	$277.88MB$	$0.4536s$	5.48%	$15.26ms$	$8.1232s$	98.13%	$8.2779s$
sha2 PPC32	$7.7194s$	92.51%	$109.19\mu s$	$281.42MB$	$0.4664s$	5.59%	$15.19ms$	$8.1854s$	98.10%	$8.3442s$

User Space Process Loader. After the OS kernel finishes its loading process, the control of the OS is given to Userspace executed processes. Every user-space process is organized in the same way; it has the same view of the available system resources, i.e. the VAS, as depicted in Figure AF1. The layout and size of the VAS is identical for every process. A typical size for the VAS is 4 GB with two segmentations: Userspace 3GB (`0x00000000 - 0x7FFFFFFF`) and Kernelspace 1GB (`0x80000000 - 0xFFFFFFFF`).

The process-loader in Linux systems is implemented by `ld-linux.so` (LD) [17]. Similar to the KML, LD loads the executable object code into SM and executes the symbol resolution and relocation process [18], if necessary (cf. Section 4.2). Usually, object code depends on shared object code provided by external libraries. Therefore, LD also loads all referenced shared libraries into memory, before the symbol resolution and relocation phase is performed. As expected, the symbol resolution and relocation is also executed for every shared library, as dependencies between different shared libraries occur very frequently. After the relocation phase of LD completed successfully, the final process image is in a ready-to-run state. In a final step, LD delegates the execution flow to the relocated object code by calling its `main()` function. From that point onward, the object code will be available as a process in the system. The final relocated process image is also depicted in Figure AF1, including the memory layout of a LD loaded library.

B.2 Memory Management: Access and Protection

MM is a core functionality, provided by the OS kernel. As yet, we briefly explained the organization of object code in ELF sections and introduced the segmentation of object code in the VAS. In the following, we will discuss MM in more detail. Therefore, we will describe the paging mechanism regarding segmentation in VAS, the memory protection schemes used to protect the system memory in modern OS, and the static and dynamic behavior of memory mapped object code components.

B.2.1 Virtual Memory Management

As described, the object code is organized in different sections within the ELF and during the loading process multiple related sections are joined to segments, representing the organizational structure in the VAS. However, the internal structure inside the OS kernel, and, furthermore, at the physical hardware layer, is organized in pages of a fixed size (usually 4096 Bytes). As a result, a VAS segment is an ordered logical representation of the physical pages mapped in memory. This additional abstraction layer between VAS and physical memory enables different process' VAS to share the same physical pages, effectively reducing the amount of required physical memory pages substantially.

In general, segments are shared between multiple processes whenever possible. However, as soon as a process writes to a shared segment, a copy-on-write mechanism in the page fault handler is executed allocating a new physical memory page for the process and copying the content of old page to the newly allocated. After that, the write-operation is executed and the newly allocated physical page is no longer shared.

Whether a VAS segment is expected to change or not, is determined during the compilation and linking phase of the ELF and depends on the individual section. In the following, the access permissions are presented in more detail and how the dynamic behavior of certain sections affects DRIVE.

B.2.2 Access Permissions of Memory Mapped Segments

Access to memory mapped segments is defined by access permission flags, controlled and enforced by the OS. Apart from security related access control mechanisms, the access permissions also determine whether a segment can be shared or not. The understanding of the access permission of segments is crucial for the DRIVE concept.

Table 1 shows the access permissions of the `.text`, `.data`, `Heap` and `Stack` VAS segments and lists their designated ELF object code sections encapsulated inside the segment, along with their individual access permissions, i.e. `(r)ead`, `(w)rite` and `e(x)ecute`. In specific circumstances, mappings with `rwx` permissions exist and are indeed necessary as, e.g., Virtual Machine and interpreter based programming languages often require access permissions considered as insecure.

DRIVE facilitates access permissions as an indicator for potential threats and therefore measures them as meta-data. The meta-data is analyzed during the verification phase and if unexpected changes to access permissions are detected the system is considered as compromised and becomes untrusted.

C. CONSTRUCTION OF MEASUREMENT, REPORTING AND VERIFICATION ARTIFACTS

Following we describe the construction of the measured and verified information, discussed in Section 4 in more detail. Table AT2 depicts the used abbreviations in the constructed data-structures.

A set (S) for a singular measured memory area, consists of individual single measurements either based on the content of the measured memory segment or section, or measured meta-data. The major difference between predictable and unpredictable S is that measured predictable segments or section include *mhd*, representing a hash digest of the measured content.

Table AT2: Abbreviations

Full Name	Abbreviation
memory start address	msa
memory size	ms
memory access permissions	map
file or module name	mf
memory hash digest	mhd

For this reason, MS for predictable segments and sections, i.e. `.text` and `.got`, are constructed as follows:

$$S_{predictable} = \{mf, msa, ms, map, mhd\}$$

In contrast to that the MS for unpredictable segments and sections do not include mhd:

$$S_{unpredictable} = \{mf, msa, ms, map\}$$

The described Measurement Sets (MS), comprise now multiple sets and combine them to a single data-structure. A single process consists of many different measurements for all different memory mapped segments and section. For instance, a `/bin/bash` process consists of over 50 memory mappings, including all shared libraries, additional opened files and other anonymous mappings. This would imply more than 50 operations for the anchoring of the measurements, which simply would not scale for a measurement of multiple processes due to the bare time consumption. Thus, a MS comprises all individual memory measurements in a single data-structure, that can be used to create a single to-be-anchored value.

$$MS = \{S_1, S_2, \ldots, S_n\}$$

This to-be-anchored value is a hash of the MS (HMS), such that:

$$\begin{aligned} HMS_0 &= digest(MS_0) \\ HMS_1 &= digest(MS_1) \\ &\cdots \\ HMS_n &= digest(MS_n) \end{aligned}$$

HMS_n now comprises the integrity of each individual $MS_{0\ldots n}$ and is extended into the SM in the same order. The extend-function for the SM anchored fingerprint ($SMAF$) is defined as follows, whereas | denotes concatenation:

$$\begin{aligned} SMAF_0 &= digest(HMS_0) \\ SMAF_1 &= digest(SMAF_0|HMS_1) \\ &\cdots \\ SMAF_n &= digest(SMAF_{n-1}|HMS_n) \end{aligned}$$

As described, SMAF is part of the SSR and used during verification to enable and integrity verification of the transmitted DML, whis is defined as:

$$DML = \{MS_0, \ldots, MS_n\}$$

Consequently, the verification constructs an Expected SMAF (ESMAF), based on the information present in the DML. The only difference is that the MS come from the transmitted DML. Accordingly the expected HMS are created:

$$\begin{aligned} EHMS_0 &= digest(MS_{DML_0}) \\ EHMS_1 &= digest(MS_{DML_1}) \\ &\cdots \\ EHMS_n &= digest(MS_{DML_n}) \end{aligned}$$

and used to create ESMAF, such that:

$$\begin{aligned} ESMAF_0 &= digest(EHMS_0) \\ ESMAF_1 &= digest(ESMAF_0|EHMS_1) \\ &\cdots \\ ESMAF_n &= digest(ESMAF_{n-1}|EHMS_n) \end{aligned}$$

As discussed, the verification process compares both values: (1) the SMAF, reported and cryptographically signed by the SM, and (2) the self calculated ESMAF. If both values are equal $SMAF_{signed} == ESMAF$, it is evidently proven that the DML was not altered and represents the measured runtime system state as seen by the verification process.

D. SYSTEM ANALYSIS FOR CODE VARIANTS

We conducted a analysis of different systems to determine the distribution of PIC and RCC code in recent Linux Distributions for different architectures. Table AT3 depicts the whole analysis that was conducted, utilizing the RV generation tool.

Table AT3: System Analysis

System Executables	all	RCC_{link}	PIE
Fedora-21.i386	687	482 (70.2)	205 (29.8)
debian-wheezy amd64	650	576 (88.6)	74 (11.4)
debian-wheezy i386	676	577 (85.4)	99 (14.6)
vivid-server amd64	808	585 (72.4)	223 (27.6)
vivid-server arm64	777	558 (71.8)	219 (28.2)
vivid-server i386	807	584 (72.4)	223 (27.6)
System Libraries	**all**	PIC	RCC_{load}
Fedora-Cloud 21.i386	921	921 (100.0)	0 (0.0)
debian-wheezy amd64	701	701 (100.0)	0 (0.0)
debian-wheezyi386	705	705 (100.0)	0 (0.0)
vivid-server amd64	892	892 (100.0)	0 (0.0)
vivid-server arm64	883	883 (100.0)	0 (0.0)
vivid-server i386	891	891 (100.0)	0 (0.0)

As discussed in Section 5.1, executables show a distribution between classic link time RCC (RCC_{link}) and PIE code variant. Depending on the analyzed distribution, this varies between 11% to 30%. The trend, considering both Fedora and Ubuntu distributions are more recent than Debian; thus, we expect a shift toward PIE code in the future even more. In general, traditional RCC does not provide a significant advantage regarding execution time; yet, it is faster because a layer of indirection is not needed. Still, the security properties that are provided through the possibility to utilize ASLR for PIE, is s strong argument to shift into this direction.

Analyzed shared libraries are indeed all PIC, as discussed. Apart from the possible utilization of ASLR, the physical memory consumption is surely the most important argument why libraries are compiled as PIC.