

# DYMO: Tracking Dynamic Code Identity

Bob Gilbert, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna

Computer Security Group  
Department of Computer Science  
University of California, Santa Barbara  
{rgilbert,kemm,chris,vigna}@cs.ucsb.edu

**Abstract.** *Code identity* is a primitive that allows an entity to recognize a known, trusted application as it executes. This primitive supports trusted computing mechanisms such as sealed storage and remote attestation. Unfortunately, there is a generally acknowledged limitation in the implementation of current code identity mechanisms in that they are fundamentally static. That is, code identity is captured at program load-time and, thus, does not reflect the dynamic nature of executing code as it changes over the course of its run-time. As a result, when a running process is altered, for example, because of an exploit or through injected, malicious code, its identity is not updated to reflect this change.

In this paper, we present DYMO, a system that provides a *dynamic* code identity primitive that tracks the run-time integrity of a process and can be used to detect code integrity attacks. To this end, a host-based component computes an *identity label* that reflects the executable memory regions of running applications (including dynamically generated code). These labels can be used by the operating system to enforce application-based access control policies. Moreover, to demonstrate a practical application of our approach, we implemented an extension to DYMO that labels network packets with information about the process that originated the traffic. Such provenance information is useful for distinguishing between legitimate and malicious activity at the network level.

**Keywords:** code identity, process integrity, access control.

## 1 Introduction

Modern operating systems implement user-based authorization for access control, thus giving processes the same access rights as the user account under which they run. This violates the *principle of least privilege* [21] because processes are implicitly given more access rights than they need, which is particularly problematic in the case of malware. A more robust strategy to mitigate the effects of running malware is to make access control decisions based on the *identity* of the executing software. That is, instead of granting the same set of privileges to all applications that are run by a user, it would be beneficial to differentiate between programs and to assign different privileges based on their individual

needs. For example, a security policy could enforce that only a particular (unmodified) word processing application should access a sensitive document, or an online banking application might refuse to carry out a transaction on behalf of a user unless it can identify that the user is executing a trusted web browser. An even stronger policy could define a set of trusted (whitelisted) applications, while the execution of any other code would be denied.

Enforcing fine-grained access control policies on an application basis requires a strong notion of *code identity* [18]. Code identity is a primitive that allows an entity (for example, a security enforcement component) to recognize a known, trusted application as it executes. Code identity is the fundamental primitive that enables trusted computing mechanisms such as sealed storage and remote attestation [20].

The state-of-the-art in implementing code identity involves taking *measurements* of a process by computing a cryptographic hash over the executable file, its load-time dependencies (libraries), and perhaps its configuration. The measurements are usually taken when a process is loaded, but just before it executes [18]. A measurement is computed at this time because it includes the contents of the entire executable file, which contains state that may change over the course of execution (e.g., the data segment). Taking a measurement after this state has been altered would make it difficult to assign a global meaning to the measurement (i.e., the code identity of the same application would appear to change).

Since the code identity primitive is fundamentally static, it fails to capture the true run-time identity of a process. Parno et al. acknowledge this limitation, and they agree that this is problematic because it makes it possible to exploit a running process without an update to the identity [18]. For example, if an attacker is able to exploit a buffer overflow vulnerability and execute arbitrary code in the context of a process, no measurement will be taken and, thus, its code identity will be the same as if it had not been exploited.

In this paper, we address the problem of static code identity, and we propose DYMO, a system that provides a *dynamic* code identity primitive that continuously tracks the run-time integrity of a process. In particular, we introduce a host-based component that binds each process to an *identity label* that implements dynamic code identity by encapsulating all of the code that the process attempts to execute. More precisely, for each process, our system computes a cryptographic hash over each executable region in the process' address space. The individual hash values are collected and associated with the corresponding process. This yields an identity label that reflects the executable code that the application can run, including dynamic changes to code regions such as the addition of libraries that are loaded at run-time or code that is generated on-the-fly, for example, by a JIT compiler or an exploit that targets a memory vulnerability.

Identity labels have a variety of practical uses. For example, labels can be used in a host-based application whitelisting solution that can terminate processes when their run-time integrity is compromised (e.g., as the result of a drive-by download attack against a web browser). Also, identity labels can enable

fine-grained access control policies such as only granting network access to specifically authorized programs (e.g., known web browsers and e-mail clients).

To demonstrate how the use of identity labels can be extended into the network, we implemented an extension to DYMO that provides provenance information to all outgoing network connections. More precisely, we extended DYMO with a component that marks each TCP connection and UDP packet with a compressed identity label that corresponds to the application code that has generated the connection (or packet). This label is embedded in the network traffic at the IP layer, and, therefore, it can be easily inspected by both network devices and by the host that receives the traffic.

We have implemented our system as a kernel extension for Windows XP and tested it on several hardware platforms (a “bare metal” installation and two virtualized environments). Our experiments show that identity labels are the same when the same application is run on different systems. Moreover, when a malware program or an exploit attempts to inject code into a legitimate application, the label for this application is correctly updated.

The contributions of this paper are the following:

- We propose a novel approach to track the run-time integrity of a process by implementing a dynamic code identity primitive. The primitive has a variety of applications, at both the OS and the network levels, to enable fine-grained access control decisions based on dynamic process integrity.
- We describe the design and implementation of DYMO, a system that extends the Windows kernel to implement the proposed integrity tracking approach.
- We demonstrate a practical application of the dynamic code identity primitive by extending DYMO to label network packets based on the application code that is the source of the traffic. This information is useful for distinguishing between legitimate and malicious activity at the network level.
- We discuss our experimental results, which show that our system is able to track dynamic process integrity in a precise and efficient manner. Moreover, we show that identity labels are robust and correctly reflect cases in which malicious code tampers with legitimate programs.

## 2 System Overview

In this section, we first discuss the requirements for our identity labels in more detail. Then, we present an overview of DYMO, our system that implements these labels and provides dynamic code identity for processes.

### 2.1 System Requirements

A system that aims to provide dynamic code identity must fulfill three key requirements: First, identity labels must be *precise*. That is, a label must uniquely identify a running application. This implies that two different applications receive different labels. Moreover, it also means that a particular application

receives the same label when executed multiple times on different hardware platforms or with slightly different dynamic libraries. This is crucial in order to write meaningful security policies that assign permissions on the basis of applications.

The second requirement is that identity labels must be *secure*. That is, it must be impossible (or very difficult) for a malicious process to assume the identity of a legitimate application. Otherwise, a malicious process can easily bypass any security enforcement mechanism that is based on code identity simply by impersonating an application that has the desired permissions.

The third requirement is that the implementation of the mechanism that computes identity labels must be *efficient*. Program execution on current operating systems is highly dynamic, and events in which a process adds additional code to its address space (typically in the form of dynamic libraries) are common. Also, the access permissions of code segments are changed surprisingly often. Thus, any mechanism that aims to maintain an up-to-date view of the running code will be invoked frequently, and, thus, must be fast.

## 2.2 System Design

To capture the dynamic identity of code, and to compute identity labels, we propose an approach that dynamically tracks all executable code regions in a process' address space. Typically, these code regions contain the instructions of the application code as well as the code sections of libraries, including those that are dynamically loaded. DYMO computes a cryptographic hash over the content of each code section, and it uses the set of hashes as the process' identity label.

*Precise Label Computation.* DYMO ensures the precision of identity labels, even in cases where an application loads slightly different sets of libraries on different executions. This can happen when applications load certain libraries only when the need arises, for example, when the user visits a web page that requires a particular browser plug-in. In such cases, two identity labels for two executions of the same application will contain an identical set of hashes for those libraries that are present in both processes, while one label will have extra hashes for any additional libraries that are loaded.

Typically, executable regions in a process' address space correspond to code sections of the binary or libraries. However, this is not always the case. For example, malicious processes can inject code into running applications (e.g., using Windows API functions such as `VirtualAllocEx` and `WriteProcessMemory`). In addition, when a legitimate application has a security vulnerability (such as a buffer overflow), it is possible to inject shellcode into the application, which alters its behavior. Our identity labels encapsulate such code, because DYMO keeps track of all executable memory regions, independent of the way in which these regions were created.

*Handling Dynamically Generated Code.* An important difference from previous systems that compute hashes of code regions to establish code identity is that DYMO supports dynamically generated code. For this, one could simply choose to hash code regions that are dynamically created (similar to regular program

code). Unfortunately, it is likely that such code regions change between program executions. For example, consider a just-in-time compiler for JavaScript that runs in a browser. Obviously, the code that is generated by this JIT compiler component depends on the web pages that the user visits. Thus, hashes associated with these code regions likely change very frequently. As a result, even though the hash would precisely capture the generated code, its value is essentially meaningless. For this reason, we decided not to hash dynamic code regions directly. Instead, whenever there are dynamically created, executable memory regions, we add information to the label that reflects the generated code and the library responsible for it. The rationale is that we want to allow only certain known (and trusted) parts of the application code to dynamically generate instructions. However, there are no restrictions on the actual instructions that these regions can contain. While this opens a small window of opportunity for an attacker, a successful exploit requires one to find a vulnerability in a library that is permitted to generate code, and this vulnerability must be such that it allows one to inject data into executable memory regions that this library has previously allocated. This makes it very difficult for a malicious program or an attacker to coerce a legitimate program to execute unwanted code.

*Secure Label Computation.* Identity labels must be secure against forging. This requires that malicious processes cannot bypass or tamper with the component that computes these labels. In other words, DYMO must execute at a higher privilege than malicious code that may tamper with the label computation.

One possible way to implement DYMO is inside a virtual machine monitor (VMM). This makes it easy to argue that the component is protected from the guest OS and non-bypassable, and it would also be a convenient location to implement our extensions, since we could use an open-source VMM. Another way to implement DYMO is as part of the operating system kernel. In this case, the threat model has to be somewhat weaker, because one must assume that malicious processes only run with regular user (non-administrator) privileges. Moreover, this venue requires more implementation effort given that there is no source code available for Windows. However, on the upside, implementing DYMO as part of the operating system kernel makes real-world deployment much more feasible, since it does not require users to run an additional, trusted layer (such as a virtual machine) underneath the OS.

For this work, we invested a substantial effort to demonstrate that the system can be implemented as part of the Windows operating system. This was a deliberate design decision that makes DYMO easier to deploy. We also believe that it is reasonable to assume that the attacker does not have root privileges. With the latest releases of its OS, Microsoft is aggressively pushing towards a model where users are no longer authenticated as administrator but run as regular users [17]. Also, recent studies have shown that malware increasingly adapts to this situation and runs properly even without administrator privileges [1].

*Efficient Label Computation.* Computing labels for programs should only incur a small performance penalty. We add only a few instructions to the fast path in the Windows memory management routines (which are executed for every page

fault). Moreover, the label computation is done incrementally; it only needs to inspect the new, executable memory regions that are added to the process address space. As a result, our label computation is fast, as demonstrated by the performance overhead measured in our experiments (which are discussed in Section 5).

### 3 System Implementation

In this section, we describe DYMO’s implementation in detail. In particular, we discuss how our system extends the Windows XP kernel to track the executable regions of a process and uses this information to compute identity labels.

Dynamically maintaining a process’ identity over the course of its execution is a difficult problem. The first concern is that processes load dynamic link libraries (DLLs) during run-time, which makes it difficult to predetermine all of the code segments that will reside in a process’ address space. Second, processes may allocate arbitrary memory regions with execute permission, for example, when dynamically generating code. This is commonly done by packed malware, which produces most of its code on-the-fly in an effort to thwart signature-based detection, but also by just-in-time compilers that generate code dynamically. A third issue concerns image rebasing. When the preferred load addresses of two DLLs conflict, one has to be relocated, and all addresses of functions and global variables must be patched in the code segment of the rebased DLL. This poses a problem because we do not want the identities of two processes to differ simply because of differences in DLL load order. DYMO is able to track a process’ identity in spite of these problems, as discussed in the following sections.

#### 3.1 System Initialization

We assume that DYMO is installed on a clean machine and is executed before any malicious process is running. Our system begins its operation by registering for kernel-provided callbacks that are associated with process creation and image loading (via `PsSetCreateProcessNotifyRoutine` and `PsSetLoadImageNotifyRoutine`, respectively) and hooking the NT kernel system services responsible for allocating memory, mapping files, and changing the protection of a memory region (these functions are `NtAllocateVirtualMemory`, `NtMapViewOfSection`, and `NtProtectVirtualMemory`, respectively).

By registering these callbacks and hooks, DYMO can observe and track all regions of memory from which a process could potentially execute code. DYMO also hooks the page fault handler so that it will be alerted when a tracked memory region has been requested for execution. This allows for the inclusion of this region into the identity label. This alert strategy makes use of hardware-enforced Data Execution Prevention (DEP/NX) [16]. DEP/NX utilizes No eXecute hardware support to disallow execute access to memory pages that have the NX bit set. Note that only those DEP/NX violations that are due to our tracking technique are processed in the hooked page fault handler. The vast majority of page faults are efficiently passed on to the original handler.

### 3.2 Identity Label Generation

An identity label encapsulates all memory regions (sets of consecutive memory pages) of a process' address space that are executed. Since each executable memory region is self-contained and can be modified independently, DYMO tracks them individually through *image hashes* and *region hashes*.

Image and region hashes are cryptographic hashes (currently we use SHA-1) that represent images (i.e., `.exe` files and DLLs) and executable memory regions, respectively. The primary difference between the two types of hashes is that the former refer to image code segments while the latter correspond to all other executable memory allocations. We make this distinction because of the differences in generating the two types of hashes, as discussed later. A basic identity label is generated by aggregating all image and region hashes into a set. In Section 4.2, we discuss an optimization step that allows us to compress the size of identity labels significantly.

Since the label is a set of hashes, the constituent image and region hashes can be individually extracted. As a result, the identity label is independent of the exact layout of executable memory regions in the process' address space (which can change between executions). Furthermore, the identity label encapsulates DLLs that are dynamically loaded according to the run-time behavior of a particular process execution (e.g., the dynamic loading of a JavaScript engine by a browser when rendering a web page that contains JavaScript). The creation of image and region hashes is described next.

*Image Hashes.* It is easiest to understand the operation of DYMO by walking through the loading and execution of an application. After a process is started and its initial thread is created – but before execution begins – DYMO is notified through the process creation callback. At this point, DYMO constructs a *process profile* to track the process throughout its execution.

Just before the initial thread starts executing, the image loading callback is invoked to notify DYMO that the application's image (the `.exe` file) and the `Ntdll.dll` library have begun loading. DYMO locates the code segment for each of these images in the process' virtual address space and modifies the page protection to remove execute access from the region. DYMO then adds the original protection (`PAGE_EXECUTE_READ`), the new protection (`PAGE_READONLY`), and the image base address to the process profile.

`Ntdll.dll` is responsible for loading all other required DLL images into the process, so the initial thread is set to execute an initialization routine in `Ntdll.dll`. Note that this marks the first user mode execution attempt in the new process. Since DYMO has removed execute access from the `Ntdll.dll` code segment, the execution attempt raises a DEP/NX exception, which results in a control transfer to the page fault handler. DYMO's page fault handler hook is invoked first, which allows it to inspect the fault. DYMO determines that this is the DEP/NX violation that it induced, and it uses the process profile to match the faulting address to the `Ntdll.dll` code segment. Using the memory region information in the process profile, DYMO creates the image hash that identifies `Ntdll.dll`. It does this by computing a cryptographic hash of the code segment.

Note that special care must be taken to ensure that the image hash is not affected by image rebasing. DYMO accomplishes this by parsing the PE header and `.reloc` section of the image file to find the rebase fixup points and revert them to their canonical values. That is, those addresses in a library's code that change depending on the library's base address are overwritten with their initial values, which are derived from the preferred base address. This is necessary to avoid the generation of different hashes when the same library is loaded at different addresses in different program executions.

The image hash is then added to the process profile. Finally, DYMO restores the original page protection (`PAGE_EXECUTE_READ`) to the faulting region and dismisses the page fault, which allows execution to continue in the `Ntdll.dll` initialization routine.

`Ntdll.dll` consults the executable's Import Address Table (IAT) to find required DLLs to load (and recursively consults these DLLs for imports) and maps them into memory. DYMO is notified of these image loads through a callback, and it carries out the processing described above for each library. The callback is also invoked when DLLs are dynamically loaded during run-time, which enables DYMO to process them as well. After loading, each DLL will attempt to execute its entry point, a DEP/NX exception will be raised, and DYMO will add an image hash for each DLL to the process profile as described above.

*Region Hashes.* Collecting image hashes allows DYMO to precisely track all of a process' loaded images. But there are other ways to introduce executable code into the address space of a process, such as creating a private memory region or file mapping. Furthermore, the page protection of any existing memory region may be modified to allow write and/or execute access.

All of these methods eventually translate to requests to one of three system services that are used for memory management – `NtAllocateVirtualMemory`, `NtMapViewOfSection`, or `NtProtectVirtualMemory` – which are hooked by DYMO. When a request to one of these system services is made, DYMO first passes it to the original routine, and then it checks whether the request resulted in execute access being granted to the specified memory region. If so, DYMO reacts as it did when handling loaded DLLs: it removes execute access from the page protection of the region, and it adds the requested protection, the granted protection, and the region base address to the process profile. When the subsequent DEP/NX exception is raised (when code in the region is executed for the first time), DYMO creates a region hash for the region. Unfortunately, generating a region hash is not as straightforward as creating an image hash (i.e., calculating a cryptographic hash over the memory region). This is because these executable regions are typically used for dynamic code generation, and so the region contents vary wildly over the course of the process' execution. Handling this problem requires additional tracking, which we describe next.

*Handling Dynamic Code Generation.* To motivate the problem created by dynamic code generation, consider the operation of the Firefox web browser. As of version 3.5, Firefox uses a component called TraceMonkey [15] as part of its JavaScript engine to JIT compile *traces* (hot paths of JavaScript code), and it

executes these traces in an allocated memory region. Since the generated code will vary depending upon many factors, it is difficult to track and identify the region (a similar issue arises with recent versions of Adobe's Flash player and other JIT compiled code). Nonetheless, care must be taken to effectively track the JIT code region as it represents a writable and executable memory region that may be the target of JIT spraying attacks [3].

To overcome this difficulty, DYMO tracks the images that are responsible for allocating, writing, and calling into the region in question. The allocator is tracked by traversing the user mode stack trace when the region is allocated until the address of the code that requested the allocation (typically a call to `VirtualAlloc`) is reached. DYMO tracks the writer by filtering write access from the region, and, in the page fault handler, capturing the address of the instruction that attempts the write. The caller is tracked by locating the return address from the call into the region. In the page fault handler, this return address can be found by following the user mode stack pointer, which is saved on the kernel stack as part of the interrupt frame. DYMO creates a (meta) region hash by concatenating the image hashes of the allocator, writer, and caller of the region and hashing the result. In the case of Firefox TraceMonkey, a hash that describes that the region belongs to its JavaScript engine housed in `Js3250.dll` is generated.

Dynamic code rewriting is handled in a similar fashion. Code rewriting occurs, for example, in the Internet Explorer 8 web browser when `Ieframe.dll` rewrites portions of `User32.dll` to detour [11] functions to its dynamically generated code region. In this case, since `User32.dll` has already been registered with the system and DYMO is able to track that `Ieframe.dll` has written to it, the `User32.dll` image hash is updated to reflect its trusted modification.

*Handling the PAGE\_EXECUTE\_READWRITE Protection.* When a process makes a call that results in a memory protection request that includes both execute and write access, DYMO must take special action. This is because DYMO must allow both accesses to remain transparent to the application. However, it must also differentiate between the two, so that it can reliably create hashes that encapsulate any changes to the region. The solution is to divide the `PAGE_EXECUTE_READWRITE` protection into `PAGE_READWRITE` and `PAGE_EXECUTE_READ` and toggle between the two.

To this end, DYMO filters the `PAGE_EXECUTE_READWRITE` request in a system service hook and, initially, only grants `PAGE_READWRITE` to the allocated region. Later, if the application attempts to execute code in the region, a DEP/NX exception is raised, and DYMO creates a hash as usual, but instead of granting the originally requested access, it grants `PAGE_EXECUTE_READ`. In other words, DYMO removes the write permission from the region so that the application cannot update the code without forcing a recomputation of the hash.

If a fault is later incurred when writing to the region, DYMO simply toggles the protection back to `PAGE_READWRITE` and dismisses the page fault. This strategy allows DYMO to compute a new hash on every execution attempt, while tracking all writes and remaining transparent to the application.

### 3.3 Establishing Identity

So far, we have described how DYMO computes the identity labels of processes. However, we have not yet discussed how these labels can be used to identify applications.

Recall that a label is a set of hashes (one for each executable memory region). One way to establish identity is to associate a specific label with an application. A process is identified as this application only when their labels are identical; that is, for each hash value in the process' label, there is a corresponding hash in the application's label. We call this the *strict matching* policy.

A limitation of the strict matching policy is that it can be overly conservative, rejecting valid labels of legitimate applications. One reason is that an application might not always load the exact same set of dynamic libraries. This can happen when a certain application feature has not been used yet, and, as a result, the code necessary for this feature has not been loaded. As another example, take the case of dynamic code generation in a web browser. When the user has not yet visited a web page that triggers this feature, the label will not contain an entry for a dynamically allocated, executable region created by the JIT compiler. To address this issue, we propose a *relaxed matching* policy that accepts a process label as belonging to a certain application when this process label contains a subset of the hashes that the application label contains *and* the hash for the main code section of the application is present.

## 4 Applications for DYMO

DYMO implements a dynamic code identity primitive. This primitive has a variety of applications, both on the local host and in the network. In this section, we first describe a scenario where DYMO is used for performing local (host-based) access control using the identity of processes. Then, we present an application where DYMO is extended to label network connections based on the program code that is the source of the traffic.

### 4.1 Application-Based Access Control

Modern operating systems typically make access control decisions based on the user ID under which a process runs. This means that a process generally has the same access rights as the logged-in user. DYMO can be used by the local host to enable the OS to make more precise access control decisions based on the identity of applications. For example, the OS could have a policy that limits network access to a set of trusted (whitelisted) applications, such as trusted web browsers and e-mail clients. Another policy could impose restrictions on which applications are allowed to access a particular sensitive file (similar to sealed storage). Because DYMO precisely tracks the dynamic identity of a process, a trusted (but vulnerable) application cannot be exploited to subvert an access control policy. In particular, when a trusted process is exploited, its identity label changes, and, thus, its permissions are implicitly taken away.

To use application-based access control, a mechanism must be in place to distribute identity labels for trusted applications, in addition to a set of permissions that are associated with these applications. The most straightforward approach for this would be to provide a global repository of labels so that all hosts that run DYMO could obtain identity labels for the same applications. We note that global distribution mechanisms already exist (such as Microsoft Update), which DYMO could take advantage of. This would work well for trusted applications that ship with Windows, and they could be equipped with default privileges.

Furthermore, it is also straightforward for an administrator to produce a whitelist of identity labels for applications that users are allowed to run, for example, in an enterprise network. To this end, one simply needs to run an application on a system where DYMO is installed, exercising the main functionalities so that all dynamic libraries are loaded. The identity label that our system computes for this application can then be readily used and distributed to all machines in the network. In this scenario, an administrator can restrict the execution of applications to only those that have their labels in a whitelist, or specific permissions can be enabled on a per-application basis.

One may argue that during this training period it may not be feasible to fully exercise an application so as to guarantee that all possible dynamic libraries are loaded. The problem is that, after DYMO is deployed, untrained paths of execution could lead an application to load unknown libraries that would invalidate the application's identity label, resulting in a false positive. We believe that such problems can be mitigated by focused training that is guided by the users' intended workflow. Furthermore, an administrator may accept a small number of false positives as a trade-off against spending more time to reveal an application's esoteric functionality that is rarely used.

## 4.2 DYMO Network Extension

In this section, we describe our implementation of an extension to DYMO to inject a process' identity label into the network packets that it sends. This allows network entities to learn the provenance of the traffic. An example scenario that could benefit from such information is an enterprise deployment.

In a homogeneous enterprise network, most machines will run the same operating system with identical patch levels. Moreover, a centralized authority can enforce the software packages that are permissible on users' machines. In this scenario, it is easy to obtain the labels for those applications and corresponding libraries that are allowed to connect to the outside Internet (e.g., simply by running these applications under DYMO and recording the labels that are observed). These labels then serve as a whitelist, and they can be deployed at the network egress points (e.g., the gateway). Whenever traffic with an invalid label is detected, the connection is terminated, and the source host can be inspected.

By analyzing labels in the network, policies can be enforced at the gateway, instead of at each individual host, which makes policy management simpler and more efficient. Furthermore, the DYMO network extension allows for other traffic monitoring possibilities, such as rate limiting packets from certain applications

or gathering statistics pertaining to the applications that are responsible for sending traffic through the network.

To demonstrate how identity labels can be used in the network, we implemented the DYMO network extension as a kernel module that intercepts outbound network traffic to inject all packets with the identity label of the originating process. We accomplish this by injecting a custom IP option into the IP header of each packet, which makes it easy for network devices or hosts along the path to analyze the label. In addition, as an optimization, the label is only injected into the first packet(s) of a TCP connection (i.e., the SYN packet).

The *injector*, a component that is positioned between the TCP/IP transport driver and the network adapter, does the injection to ensure that all traffic is labeled. A second component, called the *broker*, obtains the appropriate identity label for the injector. These components are discussed next.

**The Injector.** The injector component is implemented as a Network Driver Interface Specification (NDIS) Intermediate Filter driver. It sits between the TCP/IP Transport Provider (`Tcpip.sys`) and the network adapter, which allows it to intercept all IP network traffic leaving the host. Due to the NDIS architecture, the injection component executes in an arbitrary thread context. Practically speaking, this means that the injector cannot reliably determine on its own which process is responsible for a particular network packet. To solve this problem, the injector enlists the help of a broker component (discussed below).

When a packet is passed down to the injector, it inspects the packet headers and builds a *connection ID* consisting of the source and destination IP addresses, the source and destination ports, and the protocol. The injector queries the broker with the connection ID and receives back a process identity label. The label is injected into the outbound packet as a custom IP option, the appropriate IP headers are updated (e.g., header length and checksum), and the packet is forwarded down to the network adapter for delivery.

**The Broker.** The broker component assists the injector in obtaining appropriate identity labels. The broker receives a connection ID from the injector and maps it to the ascribed process. It then obtains the label associated with the given process and returns it to the injector.

The broker is implemented as a Transport Driver Interface (TDI) Filter driver. It resides above `Tcpip.sys` in the transport protocol stack and filters the TDI interfaces used to send packets. Through these interfaces, the broker is notified when a process sends network traffic, and it parses the request for its connection ID. Since the broker executes in the context of the process sending the network traffic, it can maintain a table that maps connection IDs to the corresponding processes.

**Label Size Optimization.** Identity labels, which store all image and region hashes for a process, can become large. In fact, they might grow too large to fit into the IP option field of one, or a few, network packets. For example, consider the execution of Firefox. It is represented by 87 image and region hashes, each

of which is a 20 byte hash value, which results in an identity label size of 1.74 KB. To compress identity labels before embedding them into network packets, DYMO uses Huffman encoding to condense image and region hashes into image and region codes. DYMO then simply concatenates the resulting image and region codes to generate the label that is sent over the network.

The Huffman codes are precomputed from a global input set which includes all trusted applications and DLLs (with their different versions), with shorter codes being assigned to more popular (more frequently executed) images. The codes are stored in a lookup table when DYMO begins operation. To generate a Huffman code for an image hash, the system uses the computed hash of the image to index into the lookup table and obtain the corresponding Huffman code. If the lookup fails, DYMO generates an `UNKNOWN IMAGE` code to describe the image; thus, untrusted or malicious images are easily detected. To generate a region code, DYMO uses the hashes of the allocator, writer, and caller of the region to compute a hash to index into the lookup table. If the lookup fails, DYMO generates an `UNKNOWN REGION` code to describe the region.

In the current implementation, Huffman codes vary in length from 6 to 16 bits. When using optimized codes, DYMO generates an identity label for Firefox that is 74 bytes, which is 4.25% of its size in the unoptimized case. Note that the maximum size of the IP option is fixed at 40 bytes. For identity labels that exceed this 40 byte limit, we split the label over multiple packets.

## 5 Evaluation

We evaluated DYMO on three criteria that address the system requirements discussed in Section 2.1: the precision of the identity labels it creates, its ability to correctly reflect changes to the identity label when a process has been tampered with, and its impact on application performance.

### 5.1 Label Precision

In order for an identity label to be meaningful, it must uniquely identify the running application that it represents. That is to say, two different applications should receive different labels, and the same application should receive the same label when it is executed multiple times on the same or different hosts. We say that a label meeting these criteria is *precise*.

To evaluate the precision of DYMO's identity labels, we deployed the Windows XP SP3 operating system on three different platforms: a virtual machine running under VMware Fusion 2 on a Mac OS X host, a virtual machine running under VirtualBox 3.1 on an Ubuntu host, and a standard, native installation on bare metal. We then created a test application suite of 107 executables taken from the Windows `System32` directory. To conduct the experiment, we first obtained our database of identity labels using the training method described in Section 4.1, that is, by simply running the applications on the test platforms and storing the resulting labels. We then ran each application from the test suite on every platform for ten seconds and for three iterations. In addition, we

performed similar tests for Internet Explorer, Firefox, and Thunderbird, which are examples of large and complex applications. For these programs, instead of only running the applications for ten seconds, we simulated a typical workflow that involved browsing through a set of websites – including sites containing JavaScript and Flash content – with Internet Explorer and Firefox and performing mail tasks in Thunderbird.

We found that in all cases, the generated identity labels were precise. There were small differences in the dynamic loading of a few DLLs in some of the processes, but according to the relaxed matching policy for establishing identity as described in Section 3.3, all processes were accepted as belonging to their corresponding applications. More specifically, for 99 of the 107 programs (93%), as well as for Firefox and Thunderbird, the generated labels were identical on all three platforms. In all other cases, the labels were identical among the three runs, but sometimes differed between the different platforms. The reason for the minor differences among the labels was that a particular library was not present (or not loaded) on all platforms. As a result, the applications loaded a different number of libraries, which led to different labels. For six programs, the problem was that the native host was missing an audio driver, and our test suite contained several audio-related programs such as `Mplay32.exe`, `Sndrec32.exe`, and `Sndvo132.exe`. In one case, the VirtualBox platform was missing DLLs for AppleTalk support. In the final two cases (`Magnify.exe` and Internet Explorer), the VirtualBox environment did not load `Msvcp60.dll`.

Our experiments demonstrate that identity labels are precise across platforms according to the relaxed matching policy. In some special cases, certain libraries are not present, but their absence does not change the fundamental identity of the application.

## 5.2 Effect of Process Tampering

An identity label encodes the execution history of a process. We can leverage this property for detecting suspicious behavior of otherwise benign processes when they are tampered with by malware or exploits.

**Tampering by Malware.** We identified three malware samples that perform injection of code into the address space of other running processes. The first sample was a Zeus bot that modified a running instance of Internet Explorer by injecting code into `Browseui.dll` and `Ws2help.dll`. The second sample was a Korgo worm that injected a remote thread into Windows Explorer and loaded 19 DLLs for scanning activity and communication with a Command and Control (C&C) server. The third sample was a suspicious program called YGB Hack Time that was detected by 33 out of 42 (79%) antivirus engines in VirusTotal. YGB injected a DLL called `Itrack.dll` into most running processes, including Internet Explorer.

We executed the three samples on a virtual machine with DYMO running. The identity labels of the target applications changed after all three malware samples were executed and performed their injection. This demonstrates that DYMO is

able to dynamically update a process' identity label according to changes in its execution.

**Tampering by Exploits.** An alternative way to tamper with a process' execution is through an exploit that targets a vulnerability in the process. Two common attack vectors are the buffer overflow exploit and drive-by download attack. To demonstrate DYMO's ability to detect such attacks, we used the Metasploit Framework to deploy a VNC server that targets a buffer overflow vulnerability in RealVNC Client and a web server to simulate the Operation Aurora drive-by download exploit [24]. For both attacks, we configured Metasploit to use a sophisticated Reflective DLL Injection exploit payload [5] that allows a DLL to load itself into the target address space without using the facilities of the `Ntdll.dll` image loader. This makes the injection stealthier because the DLL is not registered with the hosting process (e.g., the DLL does not appear in the list of loaded modules in the Process Environment Block).

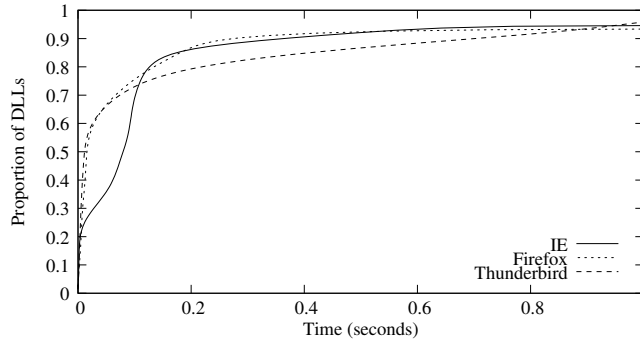
We deployed our attack VNC server and web server and navigated to them using a vulnerable version of RealVNC Client and Internet Explorer, respectively. The identity labels changed for both vulnerable applications after the attack because of the execution of code in RealVNC Client's stack, Internet Explorer's heap, and the DLL injected into the address space of both. This demonstrates that DYMO is able to update a process' identity label even in the face of a sophisticated attack technique designed to hide its presence.

### 5.3 Performance Impact

DYMO operates at a low level in the Windows XP kernel and must track when a process loads DLLs and makes memory allocation or protection change requests. Moreover, the system adds some logic to the page fault handler. Since these kernel functions are frequently invoked, care must be taken to maintain an acceptable level of performance.

Typically, a process will perform most, if not all, of the code loading work very early in its lifetime. Figure 1 shows an example of DLL loading over time for Internet Explorer, Firefox, and Thunderbird (only load-time DLLs are included). Note that 95%, 93%, and 97% of the DLLs were loaded within one second after launching Internet Explorer, Firefox, and Thunderbird, respectively.

The loading of DLLs results in the most work (and overhead) for DYMO, because it means that the system has to compute hashes for new code pages. Thus, the overhead during startup constitutes a worst case. To measure the startup overhead, we ran Internet Explorer, Firefox, and Thunderbird on the native platform, and we measured the time until each application's main window responded to user input with and without DYMO. We used the PassMark AppTimer tool to do these measurements. Table 1 shows the results. It can be seen that, with our system running, the startup times for Internet Explorer, Firefox, and Thunderbird increased by 80%, 41%, and 31%, respectively. While the overhead for Internet Explorer seems high at first glance, the browser still starts in less than one second. We feel that this is below the threshold of user awareness; therefore,



**Fig. 1.** DLL loading over time

it is an acceptable overhead. We speculate that the higher overhead of Internet Explorer can be attributed to its multi-process, Loosely-Coupled IE (LCIE) architecture [23], which results in DYMO duplicating its initialization efforts over the frame and tab processes.

**Table 1.** Startup times (in milliseconds)

Application	Without DYMO	With DYMO	Overhead
Internet Explorer	447	804	80%
Firefox	450	634	41%
Thunderbird	799	1047	31%

In addition to the worst-case overhead during application startup, we were also interested in understanding the performance penalty due to our modifications to the memory management routines and, in particular, the page fault handler. To this end, we wrote a tool that first allocated a 2 GB buffer in memory and then stepped through this buffer, touching a byte on each consecutive page. This caused many page faults, and, as a result, it allowed us to measure the overhead that a memory-intensive application might experience once the code regions (binary image and libraries) are loaded and the appropriate identity label is computed. We ran this test for 20 iterations and found that DYMO incurs a modest overhead of 7.09% on average.

## 6 Security Analysis

In this section, we discuss the security of our proposed identity label mechanism. In our threat model, we assume that the attacker controls a malicious process and wants to carry out a security sensitive operation that is restricted to a set of applications with known, trusted identities (labels). Similarly, the attacker might want to send a network packet with the label of a trusted process.

The malicious process could attempt to obtain one of the trusted labels. To this end, the attacker would have to create executable memory regions that hash to the same values as the memory regions of a trusted process. Because we use a strong hash function (SHA-1), it is infeasible for the attacker to allocate an executable region that hashes to a known value. It is also not possible to simply add code to a trusted program in order to carry out a sensitive operation on the attacker’s behalf (a kind of confused deputy attack [10]). The reason is that any added executable region would contribute an additional, unknown hash value to the identity label, thereby invalidating it.

A malware process could also attempt to tamper with the data of a process and indirectly modify its operations so that it could carry out malicious activity. This is a more difficult attack, and its success depends on the normal functionality that is implemented by the targeted victim program. The easiest way to carry out this attack is via a debugger, which allows easy manipulation of the heap or stack areas of the victim application. We prevent this attack by disabling access to the Windows debugging API for all user processes when our system is running. We believe that these APIs are only rarely used by regular users, and it is reasonable to accept the reduced functionality for non-developers.

Another way to tamper with the execution of an application without injecting additional code is via non-control-data attacks. These attacks modify “decision-making data” that might be used by the application while carrying out its computations and interactions. Previous work [4] has shown that these attacks are “realistic threats,” but they are significantly more difficult to perform than attacks in which arbitrary code can be injected. Moreover, for these attacks to be successful, the malware has to find an application vulnerability that can be exploited, and this vulnerability must be suitable to coerce the program to run the functionality that is intended by the malware author. Our current system does not specifically defend against these attacks. However, there are a number of operating system improvements that make exploits such as these significantly more difficult to launch. For example, address space layout randomization (ASLR) [2] provides a strong defense against attacks that leverage return-oriented programming (advanced return-into-libc exploits) [22]. Because our technique is compatible with ASLR, our system directly benefits from it and will likely also profit from other OS defenses. This makes this class of attacks less of a concern.

## 7 Related Work

The goal of our system is to track the run-time identity of executing processes. This objective is related to previous contributions that focus on identifying local and remote applications.

**Local Identification.** Patagonix [14] is a hypervisor-based system that tracks all executing binaries on a host with the goal of detecting the presence of processes that may be hidden by a rootkit. The system runs the target host in a virtual machine and provides a secure channel to identify and list the host’s running processes in a separate trusted VM.

The technique used by Patagonix to identify executing processes is similar to ours in that both systems leverage NX hardware support to detect code execution. However, there are some disadvantages to the Patagonix approach: First, the hypervisor must bridge a semantic gap. For example, it cannot determine when processes terminate or when requests are made to change page permissions. To combat this, the system periodically refreshes its state by remarking all pages as non-executable. This adds more overhead as all subsequent executions of pages that are already monitored will induce spurious page faults that will have to be checked. Clearly, there is a trade-off between this overhead and the fidelity of Patagonix’s view of the current state of the operating system. Furthermore, the refresh interval offers a potential vulnerability to attack. Second, Patagonix does not support JIT compiled code. It can detect and report the presence of the JIT engine, but it ignores the JIT code itself. In contrast, DYMO handles these issues.

The problems with static code identity that we have described are closely related to those surrounding data integrity tools, such as Tripwire [12]. This has led to the development of various program-level anomaly detection systems that focus on characterizing application behavior, typically by monitoring system calls [6] and their arguments [13]. Likewise, work in the area of digital rights management (DRM) has recognized how brittle static hashing is for content identification purposes, and so more robust hashing mechanisms have been proposed [8].

**Remote Identification.** Sailer et al. present an approach to integrity measurement that uses a Trusted Platform Module (TPM) to identify applications for remote attestation [20]. The hashes are computed at application load-time, so the identity measurements are fundamentally static. DYMO, on the other hand, implements a dynamic code identity primitive that also measures changes to the process during run-time. Haldar et al. argue that traditional remote attestation techniques attest to the (static) identity of a binary, when, in fact, it is an attestation to the application’s *behavior* that is desired. Their proposal, semantic remote attestation [9], is complementary to ours.

Network access control systems regulate hosts’ access to the network by ensuring that they abide by a given policy (e.g., the hosts are fully patched and are running updated antivirus software). Policies are enforced either by agents on the hosts themselves or in the network [7].

Pedigree [19] is an example of a distributed information flow tracking system that uses taint sets to record interactions between processes and resources, and it attaches these taint sets to network packets in order to exchange information between hosts. Distributed information flow tracking systems are related to our network extension to DYMO, but the semantics of labels is different.

## 8 Conclusions

This paper presents DYMO, a system that provides a dynamic code identity primitive that enables tracking of the run-time integrity of a process. Our system

deploys a host-based monitoring component to ensure that all code that is associated with the execution of an application is reliably tracked. By dynamically monitoring the identity of a process in a trustworthy fashion, DYMO enables an operating system to enforce precise application-based access control policies, such as malware detection, application whitelisting, and providing different levels of service to different applications. In addition, we implemented an application that extends DYMO so that network packets are labeled with information that allows one to determine which program is responsible for the generation of the traffic. We have developed a prototype of our approach for the Windows XP operating system, and we have evaluated it in a number of realistic settings. The results show that our system is able to reliably track the identity of an application while incurring an acceptable performance overhead. Future work will focus on extending this approach to other platforms (such as Linux) and on developing sophisticated network-level policy enforcement mechanisms that take advantage of our identity labels.

**Acknowledgments.** This work was partially supported by ONR grant N0001-40911042, ARO grant W911NF0910553, NSF grants CNS-0845559 and CNS-0-905537, and Secure Business Austria.

## References

1. Bayer, U., Habibi, I., Balzarotti, D., Kirida, E., Kruegel, C.: A View on Current Malware Behaviors. In: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (2009)
2. Bhatkar, S., DuVarney, D., Sekar, R.: Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: 12th USENIX Security Symposium (2003)
3. Blazakis, D.: Interpreter Exploitation. In: 4th USENIX Workshop on Offensive Technologies (2010)
4. Chen, C., Xu, J., Sezer, E., Gauriar, P., Iyer, R.: Non-Control-Data Attacks Are Realistic Threats. In: 14th USENIX Security Symposium (2005)
5. Fewer, S.: Reflective DLL Injection. Tech. rep., Harmony Security (2008)
6. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for UNIX Processes. In: 17th IEEE Symposium on Security and Privacy (1996)
7. Frias-Martinez, V., Sherrick, J., Stolfo, S.J., Keromytis, A.D.: A Network Access Control Mechanism Based on Behavior Profiles. In: 25th Annual Computer Security Applications Conference (2009)
8. Haitzma, J., Kalker, T., Oostveen, J.: Robust Audio Hashing for Content Identification. In: 2nd International Workshop on Content-Based Multimedia Indexing (2001)
9. Haldar, V., Chandra, D., Franz, M.: Semantic Remote Attestation A Virtual Machine Directed Approach to Trusted Computing. In: 3rd USENIX Virtual Machine Research and Technology Symposium (2004)
10. Hardy, N.: The Confused Deputy. *Operating Systems Review* 22(4), 36–38 (1988)
11. Hunt, G., Brubacher, D.: Detours: Binary Interception of Win32 Functions. In: 3rd USENIX Windows NT Symposium (1999)

12. Kim, G.H., Spafford, E.H.: The Design and Implementation of Tripwire: A File System Integrity Checker. In: 2nd ACM Conference on Computer and Communications Security (1994)
13. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the Detection of Anomalous System Call Arguments. In: Sneekenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003)
14. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor Support for Identifying Covertly Executing Binaries. In: 17th USENIX Security Symposium (2008)
15. Mandelin, D.: An Overview of TraceMonkey (July 2009), <http://hacks.mozilla.org/2009/07/tracemonkey-overview/>
16. Microsoft Corporation: A detailed description of the Data Execution Prevention (DEP) feature (September 2006), <http://support.microsoft.com/kb/875352>
17. Microsoft Corporation: Windows Vista Application Development Requirements for User Account Control (UAC) (April 2007), <http://msdn.microsoft.com/en-us/library/aa905330.aspx>
18. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping Trust in Commodity Computers. In: 31st IEEE Symposium on Security and Privacy (2010)
19. Ramachandran, A., Bhandankar, K., Tariq, M.B., Feamster, N.: Packets with Provenance. Tech. Rep. GT-CS-08-02, Georgia Institute of Technology (2008)
20. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: 13th USENIX Security Symposium (2004)
21. Saltzer, J.H., Schroeder, M.D.: The Protection of Information in Computer Systems. *Proceedings of the IEEE* 63(9), 1278–1308 (1975)
22. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: 14th ACM Conference on Computer and Communications Security (2007)
23. Zeigler, A.: IE8 and Loosely-Coupled IE (LCIE) (March 2008), <http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx>
24. Zetter, K.: Google Hack Attack Was Ultra Sophisticated, New Details Show (January 2010), <http://www.wired.com/threatlevel/2010/01/operation-aurora/>