Provenance based Integrity Protection for Windows

***Abstract*—Existing malware defenses are primarily reactive in nature, with defenses effective on malware that has been widely observed. Unfortunately, we are witnessing a generation of stealthy, highly targeted exploits and malware that today's malware defenses are unprepared for. This requires new defenses that are, by design, secure against unknown malware. In this paper, we present an approach that provides systematic defense against malware by tracking the origin of code and data on the system, and ensuring that any application that is influenced by untrusted code/data will be sandboxed. Instead of trying to protect individual applications, our approach focuses on the overall system integrity. We present a design for system-wide provenance tracking, and an approach for sandboxing untrusted (and hence potentially malicious) actors. A key benefit of our approach is that it is designed for Windows, the most widely deployed desktop OS, and the primary platform targeted by malware. It is compatible with versions of Windows from Windows XP to Windows 8.1, and supports a wide range of feature rich applications including all popular browsers, Office software, media players, Acrobat, and so on. It imposes minimal performance overheads, while being able to stop a variety of malware attacks, including the Sandworm vulnerability reported last month. We will make SPIF available as open-source software by the time of this conference.**

I. Introduction

The number of malware strains has skyrocketed in recent years. According to PandaLabs, they detected 30 million new malware strains in circulation for the year 2013 [1]. In the first quarter of 2014 [14], they have already detected 15 million new malware strains. Not only the rate of appearance of new malware has doubled, malware is also becoming more sophisticated and stealthy. Some of these stealthy strains are being employed by nation states as well as criminal enterprises in activities ranging from industrial espionage to large-scale theft of financial data.

Existing malware defenses are mainly reactive. They rely on analyzing and fingerprinting new malware in the lab, and using these fingerprints to identify malware on end systems. These defenses are effective against malware that is widely distributed, but malware that is targeted at just a few individuals may never be spotted and analyzed. Moreover, fingerprinting techniques have a hard time keeping up with the rate at which new malware strains are evolving.

A more principled alternative against unknown malware is sandboxing, exemplified by the Apps on mobile OSes. Mobile operating systems like Android, as well as recent versions of Windows and Mac OS support this form of sandboxing. Each app lives inside a container environment and its visibility is limited to this container. By default, each app is isolated from all others — the data and code of an app is not visible to other apps. Strict isolation can prevent a malicious app from compromising other apps. However, such isolation also prevents them from working together is ways that are helpful to users. In order to allow such cooperation, these OSes provide mechanisms for sharing information. These mechanisms can, unfortunately, be used by malware to compromise or contaminate other apps. Furthermore, the app environment requires applications to be self-contained, and generally imposes stricter limitations than typical desktop environments. As a result, most legacy and feature-rich applications like Photoshop and other conventional office software do not run as apps.

Another emerging defense is to refactor application code and leverage existing sandboxing mechanisms within OSes, as exemplified by the Chromium browser [4]. Applications have to be refactored into two components: brokers and workers. The worker components handle untrusted and potentially malicious data, and hence can be compromised. Workers are run inside OS sandboxes, restricting their interactions to only a small and well-defined set of interfaces, e.g., Google Chrome workers can only access CPU and memory, but not the file system. Broker components, which run as higher privilege, then mediate and grant accesses for workers to function. As brokers only expose limited interfaces to workers, the attack surface is significantly reduced.

Different OSes provide different sandboxing mechanisms. Windows (Vista and later versions) incorporates integrity level as a mandatory access control mechanism to protect higher integrity processes from lower integrity processes. It enforces no-write-up policies on various operations (e.g., a low integrity process cannot write high integrity files and registry, or memory of a high integrity process). Unix provides seccomp to confine system calls that processes can make. A typical policy is to allow untrusted processes to only perform `exit`, `sigreturn`, `read` and `write` system calls. These mechanisms prevent sandboxed processes from opening files directly.

Broker architecture has seen limited adoption because it requires developers to re-architect their applications in order use sandboxes. While a handful of applications like Chrome, Internet Explorer, Microsoft Office, and Adobe Reader have adopted this defense, a lot of the existing applications have not, and do not have incentives to re-architect. Second, these sandboxes are OS-dependent. Different applications have different ways to activate sandboxes. Applications that support multiple OSes (e.g., Chrome) have to be designed specifically for each OS. Furthermore, this architecture is not fool-proof. It reduces but does not eliminate the attack surface. All of the applications mentioned above had some vulnerabilities in their brokers, and have been successfully exploited by attackers. Finally, these sandboxes only protect against code running

inside. When code and data escape the sandboxes, they are no longer confined. A recent high-profile sandbox escape was that of Adobe Reader XI in Aug 2014, where the broker process failed to sanitize paths and hence allowed workers to create files at arbitrary locations.

A. Integrity protection by limiting information flow

As described above, sandboxing solutions have two drawbacks.

- They require significant effort from developers to rearchitect their code, or they simply won't run successfully in a sandboxed environment. This raises the question: *How do we secure the vast majority of desktop applications that simply do not run inside sandboxes?*
- Even when applications are rewritten to run inside sandboxes, the sandboxes may not be stringent enough. For instance, processes sandboxed using Windows integrity protection can still create low-integrity files, which, if subsequently consumed by a high-integrity user process, can compromise the system. This is well illustrated by the Task Scheduler XML Privilege Escalation attack [2] in Stuxnet, where the user-writable task file is maliciously modified to allow executing arbitrary commands with system privileges.

We therefore propose an alternative solution that is based on systematic tracking of the provenance of all data and code on the OS, and preventing untrusted data/code from ever influencing benign user or system processes. Unlike mechanisms that focus only on the operations of untrusted applications, our approach monitors the operations of benign applications as well, and ensures that they are not exposed to untrusted code or data. This is essential to shield potentially vulnerable benign applications from being exploited by maliciously crafted inputs.

Previous research on information flow has been focused on developing new OSes (e.g., Asbestos [6] and HiStar [18]), or building information flow on UNIX-related OSes (e.g., Flume [10], PPI [16], UMIP [11] and PIP [17]). However, we are not aware of any practical design and/or implementation of OS-wide information-flow policy enforcement for Microsoft Windows. Such a system would be highly valuable, as it would allow information flow defenses to be deployed on a platform that is the principal target of malware, and their effectiveness evaluated against real-world malware. Techniques such as PIP [17] are of limited help, because of the dearth of sophisticated malware on Linux platform. As a result, the question of effectiveness against novel forms of malware cannot be answered in a satisfactory manner.

In this paper, we describe the design of SPIF, a provenance based integrity protection system for Windows. It is architecturally similar to PIP [17], and uses the same concept of an "untrusted user." However, PIP's Unix based design differs significantly from SPIF, which is designed for Windows. The novel features of our approach include its *decoupled design,* which enables SPIF work across many versions of Windows, beginning with XP, and going all the way to Windows 8.1; and *application transparency,* which enables it to work with many complex applications such as MS Office, IE, Chrome, Firefox, Skype, Photoshop, VLC, and so on.

While numerous research works have documented information-flow and/or sandboxing based solutions for UNIX-based systems, it is rare to find papers describing similar solutions for Windows, the most common desktop OS. One of the contributions of this paper is to fill this void, and describe a design that works well for Windows OSes. Below, we describe the key features of our approach.

1) Reliable Provenance Tracking System

SPIF tracks the provenance of files and processes. While we can modify OSes to label each subject and object in the system, developing such tracking logic can be error-prone, not to mention the substantial engineering challenge. The problem is particularly serious in the context of Windows because its source code is unavailable. We therefore rely on existing security mechanisms in Windows for provenance tracking. Our design philosophy in this regard is similar to those of mobile OSes, which protect an app's data from other apps by running each of them as a different user.

OSes already support multiple users and track every file and process that users create, for auditing as well as other security purposes. SPIF re-purposes this multi-user support for reliably tracking provenance of objects as well as subjects.

Files coming from untrusted sources are owned by a new "untrusted" user. SPIF's notion of untrusted sources is compatible with Windows Security Zones — indeed, our implementation relies on zone information to automatically assign labels (i.e., whether they are benign or untrusted) to incoming files. As these file propagates on the system, any processes and files derived from untrusted files will also be labeled as untrusted.

2) Robust Policy Enforcement

Experience with various containment mechanisms, including the sandboxes used by popular browsers and applications such as Flash and Acrobat, have demonstrated the challenges of building new containment mechanisms for malicious code. One of the philosophies embraced by SPIF is to reuse existing, proven enforcement mechanisms rather than develop new ones. While provenance tracking is based on one half of multi-user protection, namely, managing security credentials of a subject, SPIF's enforcement relies on the other half, namely, enforcing file and other OS-level permissions. By relying on a mature protection mechanism that was designed into the OS right from the beginning, and has withstood decades of efforts to find and exploit vulnerabilities, SPIF side-steps the challenges of securely confining malicious code.

3) Application and OS Transparency

Existing broker architectures, which require applications to be rewritten, and moreover, require OS-specific sandboxing mechanisms. In contrast, SPIF requires neither. As a result,

SPIF can support many large applications such as Photoshop, Microsoft Office, Adobe Reader, Windows Media Player, Internet Explorer, and Firefox.

Our approach is not focused on any single application, but the overall system integrity. Instead of protecting each and every single application from getting compromised, which will require in-depth knowledge about each application, we treat applications as black boxes and assume they are vulnerable. System integrity no longer depends on the most vulnerable and exposed applications, but simply the OS to enforce access controls properly. Whenever processes consume untrusted files, they can potentially be controlled by attackers and behave maliciously. Hence, files from potentially untrusted sources are given untrusted labels (i.e., owned by the userid *untrusted*) to indicate their potentially malicious intentions. Files and processes derived from these files will inherit these untrusted labels. Existing multi-user protections are configured to ensure that untrusted processes (i.e., processes running with the userid of *untrusted*) cannot overwrite system files or the files of other users, nor can they compromise processes belonging to other users. We rely on these mechanisms to confine untrusted code and data (i.e., potentially malware) from compromising the system.

To ensure integrity, it is not sufficient to confine just the untrusted processes. It is also necessary to protect *benign processes,* which are processes run by normal users and/or the system, from being exposed to untrusted data or code. SPIF relies on intercepting security-relevant system APIs, and blocking operations made by benign processes that could result in interactions with untrusted processes, or consuming untrusted files. Note that intercepting APIs is not a secure option for untrusted code, since such code may circumvent interception, e.g., by invoking system calls directly, or by tampering with the interception mechanism. However, benign processes are those that are running code from trustworthy sources, and hence have no incentive to evade enforcement mechanisms. Note that we do not have to worry about exploits either: exploits too require interaction with untrusted sources, such as opening an untrusted file. SPIF will block such interactions, and hence can ensure that benign processes have not been subverted. As a result, our interception mechanisms are appropriate for securing benign processes.

4) Usable Policy

One of the design goal of SPIF is to be usable while being secure. Unprotected systems do not confine interactions between subjects and objects. While this allows maximum compatibility with existing software, it also allows malware to compromise system integrity. Preventing such compromise requires placing some restrictions on the operations of benign as well as untrusted applications. However, simply blocking all such operations can lead to application failures, and hence can have an undesirable impact on system usability.

We have designed SPIF with the intent of minimizing its impact on usability. Our design in this regard is based on inferring the intent of users, and accomplishing it in a manner that protects system integrity. For example, suppose that the user double-clicks on an untrusted document. Instead of denying a benign document reader from opening the untrusted document, SPIF will start the document reader as untrusted so that it can open the document. This is safe because the document reader will now be confined, and prevented from interacting with any other benign process, while its outputs are labeled as untrusted. In the same example, the document reader may try to modify its preference file. In this operation, an untrusted process is trying to write a benign file, and hence should not be permitted. Instead of denying the operation, which can potentially cause the document reader to crash, SPIF transparently redirects the request to a shadowed version of the preference file that is owned by the untrusted user.

5) Implementation on Windows

We developed SPIF on Windows, supporting Windows XP, Windows 7 and Windows 8.1. Implementing such a system-wide provenance tracking system on closed-source OSes is challenging. We share our experiences and lessons on implementing SPIF on Windows.

Research efforts in developing security defenses have been centered on Unix systems. Prototypes are developed and evaluated on open source platforms like Linux or BSD to illustrate feasibility and effectiveness of defenses. While these open source platforms simplify prototype development, they do not mirror closed source operating systems like Windows. First, these closed source operating systems are far more popular among end-users. They attract not only application developers, but also malware writers. Second, there is only a limited exposition on the internals of closed source operating systems, and hence very few researchers are aware of how the mechanisms provided in these OSes can be utilized to build systems that are secure, scalable, and compatible with large applications. For this reason, we believe the design and experience presented in this paper is valuable. To be helpful to a broad audience, the description of our system is provided in terms of concepts and features of Unix. We hope this will enable more of the systems community that is rooted in Unix to develop solutions for commercial OSes, where far more vulnerabilities are being exploited in the wild.

We developed several techniques to optimize the performance of SPIF. We showed that SPIF has low overhead performance and is effective against a wide-range of malware.

We will open-source SPIF by the time of this conference.

II. Threat Model

We assume that file origins can be partitioned into: *benign* and *untrusted*. This classification can be understood in terms of (potential) intent to compromise system integrity. Benign entities have no intent to (and do not attempt to) evade defensive mechanisms, or compromise the OS or any other application. In contrast, some (or all) of the untrusted entities can be malicious — they may intentionally subvert enforcement mechanisms, attempt to compromise other applications, processes, files, and so on.

Users of a system protected by SPIF are assumed to be benign. As a result, any benign application invoked by a user will be non-malicious.

We assume that any files received from unknown or untrusted sources will be labeled as untrusted. In practice, this can be achieved by exclusion: only verified files from trusted sources like OS distributors and trustworthy developers/vendors are labeled as benign. Other files are labeled as untrusted. As we describe later, SPIF's labeling of incoming files is smoothly coupled with Windows Security Zones that would be familiar to IE users.

We also assume that applications are vulnerable. Applications consuming untrusted resources can be compromised and controlled by attackers. Attacks can take multiple steps and involve multiple applications. The goal of the attack is to compromise the system integrity, i.e., perform modifications to the system that were not intended by the user, and enable the malware to subvert other applications or the OS.

Traditional OS mechanisms do not distinguish between user intentions, and the intents of the programs they run. This inability is exploited by malware to compromise the system. SPIF relies on secure tracking of provenance to ensure that all potentially malicious entities, i.e., entities whose content has been influenced in some way by untrusted sources, are labeled as untrusted. This enables SPIF to recognize all instances when the intent of a process could be malicious, and prevent such processes from damaging actions.

We assume that OSes provide secure multi-user isolation. Users are isolated from each another unless they explicitly grant others permissions to access their resources. We also assume that the OS kernel does not have vulnerabilities — in other words, such vulnerabilities are out of scope for this paper. We assume that benign programs rely on standard system libraries to invoke OS functions. We *do not* make such assumptions about untrusted code or processes.

In addition to protecting the system against unintended modifications, SPIF also protects benign processes from being subverted or co-opted by malware. This is achieved by providing protection to benign applications. We assume benign applications do not circumvent protection mechanisms in place. However, we do assume that malware will actively try to circumvent the protection mechanisms.

Finally, although SPIF can be configured to protect confidentiality of user files, this requires confidentiality policies to be explicitly specified, and hence we did not explore it further in this paper. It should be noted, however, that files containing secrets that can be used to gain privileges are already protected from reads by normal users, and hence they continue to be protected by SPIF.

III. Background on Windows

Security Identifiers

On Unix, processes are identified by userids and groupids. On 32-bit OSes, these ids are 32-bit integers meaningful only for the machines they belong to. Objects have user and group ownership, as well as 9-bit permission information defining read, write and execute accesses for owner, group and others. These permissions are universal across different type of objects (e.g., files, directories, IPC objects).

Windows is a more complicated system. Principals like users and groups are identified by security identifiers. They are similar to userids and groupids on Unix, but they are encode machine identifiers, making them unique and interpretable across different machines.

Security contexts of subjects and objects are encapsulated using access tokens and security descriptors respectively. Authenticated users are provided with access tokens, which contain the security identifiers of the user, as well as the groups that the user belongs to. Access tokens are like capabilities. Processes can specify using which access tokens to access resources. On Windows XP, all processes from a user with administrative privilege share the same access token that can perform administrative tasks. Windows introduced User Account Control (UAC) since Windows Vista. When an administrative user logs in, the user will be granted two access tokens: one with administrator access token, and one without. Only when the user starts administrative tasks, the applications will be given the administrator token. Other applications are only granted the normal token.

Security descriptors on objects contain ownership information, as well as access control lists (ACLs) describing accesses that security identifiers have. While there are also ACLs on Unix, Windows ACLs are far more expressive. ACLs on Unix is only limited to files. IPC objects are still not protected by file permissions. On Windows, the set of accesses that a security descriptor describes is finer grained. Apart from the basic read, write and execute permissions that Unix systems support, Windows also supports append permissions in ACLs[1] Furthermore, security descriptors are specific to object types. Different object types (e.g., files, directory, registry) each have their own set of permission settings. For example, permissions specific for files include accessing data, attributes, extended attributes and ACLs.

System API

Similar to other OSes, Windows uses system calls to allow user space applications to request services from the kernel. Like on Linux, applications typically do not invoke system calls directly, but rely on library calls. Applications invoke library calls, which then invoke system calls in the library. All OSes do not recommended applications to invoke system calls directly as they are internal to the systems. Libraries serve to hide the platform specific details. On Linux, the primarily library containing system calls is `libc`. On Windows, the library is `ntdll.dll`.

Windows system calls are also much more complicated than those on Unix. For example, a file open on Unix involves `open(2)` which takes up to 3 arguments. On the other

[1]Unix actually support append only attribute. However, such attribute is set for all users to the file. Windows ACLs allow append only operation for selected users.

hand, performing a file open operation on Windows involves `NtCreateFile`, which takes 11 arguments. Apart from the file path and open mode, additional arguments are for setting file attributes, storing request completion status, setting allocation size, controlling share access, specifying how the file is accessed, and setting extended attributes. There are 276 system calls in 32-bit Windows XP and 426 in 32-bit Windows 8.1. The number of system calls on Windows and Unix are comparable.

A Windows process can allocate and write into memory of another process owned by the same user, but not processes of another user. In the same manner, one process can create a thread in another process owned by the same user This provides a convenient way to isolate processes of different trust domains[2]. Specifically, SPIF runs untrusted processes (with potentially malicious intentions) as "untrusted" user, relying on the OSes to protect "benign users."

Registry

Apart from files, registry is another important data store on Windows. It is a hierarchical databases that store configuration settings and options about OSes and applications. While registry entries are backed by files, processes manipulate registry entries with a dedicated set of Windows APIs. Similar to files, there are permissions on each key node.

Registry entries are divided by different root keys. These root keys can be stored in a system-wide setting (e.g., `HKEY_LOCAL_MACHINE`) or user-specific preferences (e.g., `HKEY_CURRENT_USER`). Each user has her own subtree under `HKEY_USERS`, each protected with DAC permission such that only the corresponding user can access it. `HKEY_CURRENT_USER` is a symbolic link pointing to the actual subtree under `HKEY_USERS`.

Unprivileged users are given read-only access to system-wide settings, as well as read-write access to their own settings. When processes access user-specific settings, they can refer to `HKEY_CURRENT_USER` to their own settings, but `HKEY_CURRENT_USER` will be resolved by the library before actually invoking the system calls.

IV. System Design

A. Leveraging Existing Multi-user Support

SPIF provides reliable provenance tracking and robust policy enforcement. While developing our own labeling and enforcement mechanisms can yield more flexibility in policy, it can be error-prone especially when we have insufficient understanding about the OS. Because of this, we emphasize maturity and completeness of security mechanisms over factors such as flexibility or expressive power. Specifically, SPIF leverages one of the most well-studied and established security mechanisms in contemporary OSes, namely the multi-user support (DAC). The behavior of this secure multi-user isolation is well-defined. It protects against all integrity threats posed by one user's processes to another user's processes or data, as well as confidentiality threats to the extent necessary to ensure that secrets that can enable hijacking user privileges are protected. It also protects file confidentiality, as specified by file permissions.

[2]The Windows integrity protection mechanism places an additional restriction, preventing lower integrity processes from making these operations on higher integrity ones.

1) Reliable Provenance Tracking

Modern OSes support multiple users, each user has her own set of resources that are private to herself. OSes track and audit resource usage and enforce isolation policies to prevent one user from attacking another. Rich enforcement mechanisms have been developed to control how resources can be shared across multiple users. Each user has her own identifier. Every resource in the systems is encoded with access information for users and every access will be checked.

SPIF tracks provenance of files and intervenes when necessary. It transparently re-purposes the existing multi-user support to achieve reliable provenance tracking. The idea is to partition the user identity space into benign and untrusted. Subjects and objects are untrusted if they are owned by "untrusted" users. Objects can also be untrusted when they are writable by "untrusted" users. This encoding mechanism allows us to automatically reuse the reliable tracking mechanisms in OSes to track provenance. Objects like files, sockets and various IPC objects are automatically labeled as untrusted when created by untrusted processes.

For each real user R on the system, SPIF creates a *shadow user* U_R to represent untrusted subjects and objects owned by R. The "untrusted" user captures the idea that these subjects and objects may be malicious. Processes consuming these untrusted objects or communicating with these untrusted subjects can be contaminated with malicious intentions. In SPIF, only untrusted subjects can interact with untrusted subjects and objects.

On unprotected systems, every file introduced by user R belongs to R, and every process started by user R runs with R's privileges. When one of the R's files is malicious, the system cannot distinguish if the action is intended by R or not. SPIF tracks the intentions of the files and labels potentially malicious intentions using user U_R. Untrusted files introduced by user R will belong to U_R instead of R. SPIF can then distinguish if an operation is intended by the real user R, or could potentially have been hijacked. To protect user's processes, only processes owned by U_R can consume U_R's files.

2) Robust Policy Enforcement

A security mechanism should provide complete mediation against all possible attack paths. Developing such enforcement mechanisms can be tricky, especially when we are seeking a system-wide enforcement solution. Instead of developing our own enforcement, our design reuses existing security mechanisms.

SPIF uses multi-user support in OSes not only for tracking provenance, but also for robust policy enforcement against

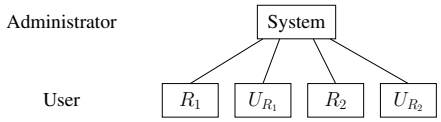

Fig. 1. Flat relationship the from OS perspective

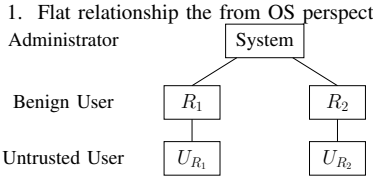

Fig. 2. Trust hierarchy of users in SPIF

malware. The same mechanism has been used to protect not only one user from another, but also for protecting system administrators against other users. Whenever a process with potentially malicious intention runs, SPIF runs it as the untrusted user U_R. From the OS perspective, this "untrusted" user is as if another user on the system. The OS enforces regular protection and makes sure that U_R cannot compromise other users, including R.

Multi-user support in OSes provides isolation, which serves as the sandbox in SPIF. While maintaining secure boundary to guard against untrusted processes, SPIF also preserves normal user experience when untrusted processes do not threaten system security. While multi-user support in OSes provides strong security protection, it does not preserve normal experience. This is because OSes treat the "untrusted" users as if completely unrelated users (a flat relationship, as in Figure 1). However, SPIF aims to achieve a hierarchical relationship between untrusted and benign users (Figure 2), allowing unidirectional information flow (one-way isolation) from benign users to untrusted. (Some of these flows can be explicitly excluded by configuring the file system to prevent U_R from reading from specified files.)

B. Relaxing the Multi-user Isolation

From the OS perspective, untrusted processes run as user U_R that has no relationship with R. However, SPIF maintains a trust relationship between U_R and R and provides U_R controlled accesses to the environment of R. As we focus on integrity, SPIF grants U_R read accesses to whatever R can read. This preserves usability when running applications as U_R. (However, the default policy of allowing U_R read access on U's files can be easily changed.) The perspective mismatch between OS and SPIF allows us to create a fail-safe environment, and explicitly grant only accesses that are safe for U_R. Accesses that are not handled by SPIF fall back to the default DAC model, which preserves integrity of the users.

1) Safe Accesses

SPIF protects system integrity against unauthorized modifications. Conventional OSes don't distinguish between the actions of a user and his/her processes. As a result, malicious or compromised application can exploit a user's privileges to carry out attacks. This is known as confused deputy attack [9]. SPIF tackles this problem by tracking provenance of information within the system. It allows systems to be modified only by processes that cannot be influenced by potentially malicious information.

SPIF allows untrusted processes of U_R to perform the following accesses:

Reading Files readable by R. SPIF focuses on integrity protection rather than secrecy. By default, every file and directory readable and accessible by R is made available to U_R. This allows SPIF to provide the same environment to untrusted processes running as U_R, as if they are running as R, as long as they do not modify R's files.

Writing to Files that R does not consume. Modifications to R's files are allowed as long as these files will not be used by R. Once files are modified by U_R, they may be contaminated with malicious intentions. If benign processes consume these files, they can be compromised. SPIF allows files to be modified only if the modifications can be shielded from R. There are two such safe cases:

- *Shadowing:* Instead of modifying R's file directly, U_R can create a shadow copy of the file and redirect all accesses to the shadow copy.
- *File creation:* Creating new files is allowed as R does not know about them, and hence can be prevented from accessing them in the future without causing failures. These new files will be owned by U_R and benign processes will not read them. If R also creates the same file, then the untrusted files will be treated as shadow files.

Apart from reading and writing to files, safe accesses can also be defined on other operations and on other objects. SPIF allows untrusted processes to list directories, execute files, query registry, rename untrusted files inside user directories, and so on. SPIF does not intervene in the operations as long as they do not directly compromise the integrity of R, or the environment that R depends on. This preserves the usability for running untrusted processes.

2) Supporting Safe Accesses

There are two ways to support safe accesses for untrusted processes:

- *Modify permissions on all objects* to grant untrusted users safe accesses, or
- *Rely on a helper process,* running with the privileges of R, to perform a subset of accesses requested by U_R

We describe and discuss these options below:

Modify permissions on files to grant safe accesses.

On Unix, file permissions are based on the 9-bit permission settings. A user file may use both owner and group permission, making it impossible to grant read-only permission to untrusted users. (Consider a user owned group writable file, but not world readable. Every bit of permission is used and cannot encode additional permission setting for the untrusted

user.) Hence, it is not possible to grant all safe accesses by modifying permissions on Unix [3]

On Windows, objects are natively protected with ACLs, which can encode arbitrary number of principals. We can grant safe accesses to all user owned objects to untrusted users. However, this will require modifying permissions on all objects of a user.

Delegation based approach for granting accesses.

Instead of modifying every object's permission settings, we can also use the delegation architecture similar to Ostia [8]. When U_R needs to access R's resource, U_R can request access to a helper process. The helper process, which runs in a trusted context, will first validate if the requested access is safe. If the access is safe, the helper process will handle the request and grant U_R the resources.

This helper process is similar to brokers in the broker architectures. However, instead of having an application specific policy to confine application behaviors, this helper process has a simple generic policy: grant only safe accesses.

Similar to Unix, most file permission checking occurs only when the files are opened. Untrusted processes whose open operation failed can can request the helper process to open the file. The helper process checks if the request is valid and safe (at this point, this just means that is is a read request), performs the open, and transfers the file handle back. On Unix, an opened file descriptor can be transferred to another process via Unix domain socket. A similar mechanism exists on Windows. File handles can be duplicated to/from another processes using `DuplicateHandle` operation. SPIF relies on this feature to grant resources to U_R, and then tells untrusted processes the file handle numbers via socket.

SPIF relies on both modifying file permissions and the delegation approach. For performance reason, it relies on ACLs to grant U_R read and execute accesses to R's files. Windows ACLs support inheritance: Files and subfolders inside can inherit ACLs from the parent folder. We can simply grant read access to the user profile directory `C:\Users` R on Windows 8.1. For other safe operations or failed read operations, we fall back to the delegation approach. This gives us the flexibility to limit what folders an untrusted process can create files. It can also support advanced policies that cannot be encoded simply using ACLs. For instance, the helper process can grant a subset of accesses to some, but not all untrusted processes. It can also transparently replace sensitive files with dummy files when sensitive files are requested.

SPIF intercepts various Windows APIs and transparently retries failed accesses with the helper process. This is achieved by dynamically hooking on entry points of some APIs. As such, failures due to DAC permissions are masked and applications are not aware of the helper process. We discuss more about our implementation in Section V.

[3]Unix does support ACLs, but this support is limited to files. More importantly, many applications are not aware of ACLs, and they can break if they rely on stat(2) for requesting ownership information.

C. Cooperative Protection for Benign Processes

Windows integrity model enforces no-write-up policy to protect higher integrity processes from being attacked by lower integrity processes. However, it does not enforce no-read-down. A higher integrity process can read lower integrity files and hence get compromised. This is well illustrated by the Task Scheduler XML Privilege Escalation attack [2] in Stuxnet, where the user-writable task file is maliciously modified to allow executing arbitrary commands with system privileges. Hence, it is important to protect benign processes from accidentally consuming untrusted resources.

While policy enforcement against untrusted processes have to be secure and tamper-resistant, protection for benign processes can be enforced in a more cooperative setting. Benign processes do not have any malicious intention and hence we assume they do not circumvent any protection mechanisms in place.

In a cooperative setting, it becomes easier to develop simpler implementation mechanisms. In particular, we are able to use mechanisms for intercepting calls to system libraries. In contrast, a non-bypassable approach will have to be implemented in the kernel, and moreover, will need to confront the problem that the actual system call API in Windows is not documented.

Similar to transparently masking failed accesses for untrusted processes, SPIF intercepts on various Windows APIs used by benign processes to protect them from consuming or interacting with untrusted entities. If the resources they operate on are untrusted, the calls will return with failures and any resources returned will be released immediately. As a result, benign processes can never consume untrusted resources and hence cannot be compromised.

V. Integration with Windows

A. Transitioning from Benign to Untrusted

Transitioning from benign to untrusted processes in Unix is relatively simple because changing privileges is a common task in Unix. A root ssh process will change its userid when a user is authenticated. Setuid operation is designed specifically to allow processes to change their userids. In Unix, a process can downgrade on exec by executing a setuid-to-root program, which will then call `setuid(2)` to change the process userid and groupid to U_R before executing the program.

Changing security identifiers on Windows relies on different mechanisms, and is in some ways trickier. On approach is to rely on a Windows primitive for changing process privileges by obtaining a different access token. A process can then impersonate the owner of the token such that access controls are checked based on that token rather than those associated with the security identifier of the process. This is typically used in system services serving requests on behalf of users. However, impersonation does not change the security identifiers of the process. Processes can voluntarily revert the privileges back to itself, making this mechanism insecure for confining untrusted processes.

Rather than changing security identifiers when programs are running, SPIF relies on changing them at process creation times. Similar to how User Account Control works, a different access token can be granted to child processes. On Windows, there exists a Windows utility `RunAs` that can be used for this purpose. This wrapper behaves like a setuid-wrapper. It does not only run programs as different users, but can also setup environment such that desktop of an untrusted process is mapped to the desktop of the actual user. When a user access her desktop running as U_R, the desktop of R will be accessed. As a result, usability is preserved. SPIF leverages this `RunAs` utility for transitioning from benign to untrusted processes.

However, a direct application of `RunAs` does not preserve usability: While the Desktop button is linked to the actual user's desktop, `RunAs` does not circumvent the multi-user isolation in the OSes. Without SPIF in place, untrusted processes are not allowed to read or create files on the users' desktop.

B. Initial File Labeling with Security Zone

An important requirement for correctly enforcing policies is to have the newly created files properly labeled based on origins. While multi-user support in OSes makes sure that files created by untrusted processes are owned by untrusted users, it does not handle files that are created as a result of downloading files using benign processes. If malicious files are mislabeled as benign, SPIF would not be able to protect system integrity.

An obvious solution is to have benign applications label files properly when they download files. This requires applications to be aware of SPIF. An alternative that is better integrated into Windows is to make use of Windows security zones that are familiar to most IE users. Security zone information is stored as Alternate Data Stream (ADS) on NTFS file system. ADS is similar to extended attributes where applications can store extra information associated with files. One particular ADS field called `Zone.Identifier` is used by Windows to associate the security zone of the files. Security zone defines the origin of the files. When users run an executable that comes from the Internet, Windows Explorer will prompt the user with a warning dialog. This information is also used by some applications to achieve better security. Applications like Microsoft Office opens a document coming from the Internet in Protected View, preventing the process itself from writing into arbitrary locations. Users have to explicitly enable editing to reopen the document without running in protected view.

On Windows, five zones are defined: `URLZONE_LOCAL_MACHINE`, `URLZONE_INTRANET`, `URLZONE_TRUSTED`, `URLZONE_INTERNET`, `URLZONE_UNTRUSTED`. These zones correspond to the trust level of the website the files are originated from. Common web browsers like Internet Explorer, Chrome or Firefox they all assign security zones when downloading files.

The origins-to-security zones mapping is customizable. Windows provides convenient user interface for users to configure what websites belong to what security zones. Additional tools are also available for enterprises to manage security zone across multiple machines with ease.

By leveraging this security zone information, files downloaded from Internet are labeled automatically. Files with `URLZONE_INTERNET` and `URLZONE_UNTRUSTED` are considered as untrusted. Unlike unprotected systems where this security zone information can easily be removed by malware, this security zone label can only be modified by benign processes. Untrusted processes in SPIF do not have permissions to remove this information since the files themselves are owned by the real user.

C. Helper Process

Helper process serves to provide safe accesses to untrusted processes, such that they can be usable while running as U_R. Ideally, the helper process only helps untrusted processes to acquire file handles. Once the file permission checking has passed, helper process can duplicate the handles to untrusted processes.

Operations on Handles.

Some operations on file handles require additional permission checking, e.g., `NtSetInformationFile` renames a file using file handle. When U_R wants to rename an untrusted file inside user directory, this call will fail. For these operations, file handles need to be duplicated back such that the operation can be performed by the helper process.

The helper process needs to check if the file handle corresponds to an untrusted file. Similar to Unix where `fstat` can retrieve ownership information from a file descriptor, Windows has `GetKernelObjectSecurity` which can retrieve security descriptors of handles. However, this call will not succeed even if the caller is a system process, unless the handles were opened with `READ_CONTROL` permission. For untrusted files opened by the helper process, they will be opened with the `READ_CONTROL` permission. The helper process can then retrieve the security descriptor for handles of untrusted files. If this information cannot be retrieved, the helper process will refuse to rename files.

Privileges of Helper Process.

Ideally, the helper process would run with the privileges of the user R to serve requests from untrusted processes U_R. This can ensure that the helper process will only be able to serve resources accessible to R. However, duplicating file handles across processes with different security identifier is a privileged operation. A helper process running with R cannot duplicate handles to U_R. Hence, the helper process runs as a system service and duplicate handle operations are always initiated by the helper process.

Since the helper process is running with the system privilege, it is important to ensure that it cannot be compromised, and all resource access operations are performed in the context of R rather than the system context. The safety of the system service is ensured by having a simple helper process, which only helps the untrusted processes to handle a few Windows operations: specifically, four file operations (`NtCreateFile`, `NtSetInformationFile`, `NtQueryFullAttributesFile` and `NtQueryAttributes`),

and a few registry operations. Specifically, the helper process helps untrusted processes to open a user's objects in read-only mode, rename untrusted objects and query user-files' attributes.

To ensure that the helper process always accesses resources using R's context, we can further decompose the helper process into two parts: a worker process running with privilege R and the broker process which runs as system. The sole purpose of the broker process is just for duplicating handles. The majority of the logic would be handled by the worker process running with privilege R. This decomposition was not used in our prototype, but there is no real difficulty in implementing it.

D. Policy Enforcement via Windows API Hooking

SPIF relies on hooking Windows APIs to provide its functionality. For benign processes, it protects them by preventing from consuming untrusted resources and get compromised. For untrusted processes, it relaxes the multi-user isolation to improve their usability. Here we describe how we achieve system-wide Windows API hooking.

Methodology. Windows signs DLLs to protect them from being tampered with. As such, we cannot modify the DLL binaries statically to hook on system calls. We instead rely on dynamic binary instrumentation tool detours [3] to hook on various Windows APIs. Detours works by rewriting function entry points with jumps to intercept handlers. Calls to the original functions will become calls to the intercepted functions. We leverage this to build wrappers around internal Windows APIs in `ntdll.dll` such that we can pre-process the API call arguments and post-process the return values.

We developed a DLL that enforces the policies for benign and untrusted processes. When loaded into process memory space, the `DLLMain` routine of the DLL will be invoked. It invokes the detours rewriting methods to perform the interception.

We rely on two methods to inject our DLL into process memory spaces. The first one is based on `AppInit_DLLs`. It is a registry key entry used by `User32.dll`. Whenever `User32.dll` is loaded into process memory spaces, the DLLs specified in the `AppInit_DLLs` registry key will also be loaded. By specifying our DLL path in `AppInit_DLLs`, our DLL will be loaded automatically when processes run.

However, `AppInit_DLLs` relies on applications to load `User32.dll`. While many GUI applications use `User32.dll`, some console applications such as the SPEC benchmarks do not rely on `User32.dll`. As such, some applications do not load our DLL. The second method that we rely on is the `CREATE_SUSPENDED` method. When creating a child process, the parent can set a flag `CREATE_SUSPENDED` to create the child process in suspended state. Once the child process is created, the parent process can create a remote thread, allocate memory and write into memory of the child process. We write the path of our DLL in the child process's memory. The remote thread can then run the `LoadLibraryA` routine with our DLL path as argument to load our DLL into the address space. Once the loading is completed, the child process is resumed if `CREATE_SUSPENDED` was set in the original parameter.

API Type	APIs
File	`NtCreateFile`, `NtOpenFile`, `NtSetInformationFile`, `NtQueryAttributes`, `NtQueryAttributesFile`,...
Process	`CreateProcess(A/W)`
Registry	`NtCreateKey`, `NtOpenKey`, `NtSetValueKey`, `NtQueryKey`, `NtQueryValueKey`,...

TABLE I
API FUNCTIONS INTERCEPTED

We rely on the first method as a way to bootstrap the API interception process. Once our library is loaded into a process, all its children process will be intercepted regardless of whether they rely on `User32.dll` or not. While this technique may miss some processes started at the early booting stage, we found that many processes (such as Windows Explorer) are intercepted.

Lower level APIs.

We are interested in mainly lower level Windows APIs in `ntdll.dll` and `kernel32.dll` because they are closer to the kernel. There are a lot of higher level Windows functions, e.g., `CreateFile(A/W)`, `FindFirstFile(A/W)`, `FindNextFile(A/W)`, `FindFirstFileEx`, `CopyFile(A/W)`, `MoveFile(A/W)`, `ReplaceFile(A/W)`, `GetProfile`... and so on. All these functions rely on a very few lower level Windows APIs such as `NtCreateFile`, `NtSetInformationFile` and `NtQueryAttributes`[4]. While working on these low level Windows internal APIs are not recommended by Microsoft, they provide a very convenient point to intercept all of the higher level functions. Our experience shows that these internal APIs do not change much from Windows XP to Windows 8.1. Furthermore, some applications like `cygwin` do not invoke higher level APIs, but invoke these internal APIs. By intercepting at the lower level, SPIF can also support these applications.

SPIF also intercepts a few higher level API functions. Higher level functions provide more semantics about call context and hence can be used for improving usability. For example, we intercept `CreateProcess(A/W)` as a way to capture user intention. If the command contains a file which is untrusted, SPIF would infer that user wants to run the untrusted file and automatically runs the command as untrusted. This is as simple as wrapping the command with the `RunAs` utility to run the new process as untrusted user. Table I shows a list of API functions that we are intercepting.

E. Handling Registry

To provide a consistent user-experience when using applications as benign and untrusted, shadowing should be applied

[4]Calls ending with "A" are for ASCII arguments, "W" are for Unicode arguments.

on registry as well. User settings in the benign applications can be accessed when the applications are used in untrusted mode. Registry shadowing should be performed as follows: If an untrusted process tries to read a registry, it should first read from its own registry. Only if such registry entry does not exist, it then attempts to read from the benign user registry. On the other hand, if an untrusted process tries to write to a registry, such writes should always be maintained in the untrusted user's registry.

However, shadowing registry accesses is more complicated than file shadowing. This is because it is very common in registry operations to open/create nodes using relative paths using multiple operations. These operations take a relative path and a root handle returned from the previous openkey/createkey operation. Ideally, based on the read/write access, SPIF can decide if shadowing is required and hence determine if a different registry path should be used. However, the use of root handle breaks registry open/create operations into multiple pieces. One has to decide which root handle should be used (whether the benign user's registry or the untrusted user's registry) without knowing what access the future operation requires. Similar to Unix, it is hard to get back the path information simply based on the file handles. Hence, SPIF has to maintain additional bookkeeping information. By maintaining this bookkeeping information, all registry open/create operations are mapped into read-only operations performed by the helper process on the benign user registry. We assumed the keys are read-only because programs tend to open registry as writable even when they only query keys. Since the handles are opened as read-only, untrusted processes cannot perform any write operations. When such writes fail, SPIF will retrieve the full path on which the operation is invoked, map it back to untrusted user such that the operation can be re-performed on the untrusted user's registry.

F. File Check Optimization

The simple 9-bit permission system on Unix makes identifying untrusted objects easy. Any process can simply perform a `stat(2)` to obtain the ownership information of the object, and find out the integrity level by examining the ownership information. If the permission settings allows untrusted users to write into it, then the object is untrusted.

Windows supports ACLs, which can encode arbitrary number of owners and groups. Each group may have multiple users. Furthermore, an user can appear in multiple entries and the final access has to be determined by some rules. This imposes extra difficulties in identifying if a file is untrusted or not.

An optimization we introduced is to avoid checking owners in the ACLs one-by- one, but to leverage the access token impersonation support in Windows. SPIF creates a new thread in every benign process of R. These new threads impersonates the untrusted user U_R and performs `AccessCheck` on writability and ownership tests on the file in question. If the impersonated thread can write into the file, or is the owner of the file, the file is considered as untrusted.

G. Caching Handle at Helper Process

SPIF improves performance by caching handles in the helper process. There are two scenarios where caching handle helps:

- While most permission checks are performed while obtaining file or registry handles, some operations do require additional permission checking. If these operations fail, the handle has to be duplicated again back to the helper process to complete the operations.
- Most registry operations operate with a root handle referring to a previously opened handle. For example, applications tend to reference to keys located in `HKEY_CURRENT_USER`. Windows library automatically caches the registry handle to `HKEY_CURRENT_USER`, and uses it as root handle for subsequent openkey operations. Instead of converting the operation to full path, SPIF can also use root handle. It is therefore beneficial to cache the handle at helper process.

SPIF maintains a fixed pool size for cached handles. These cached handles can be closed at anytime as they can be recreated when needed. The helper process maintains a LRU hash that maps $(process, remoteFileHandle)$ to $localFileHandle$. When an untrusted process request for an operation, it specifies the handle number in the request. The helper process first checks if the handle number and the process h as an entry in the LRU hash, if no cached handle is found, it will duplicate the handle. Otherwise, it will simply reuse the cached handle. A special attention is required for handle reuse. When untrusted processes close a handle h, open some resources and get the same handle number h, the helper process will have an outdated cached handle. We solve this problem by having untrusted processes to invalidate the cached handle in helper process. Messing the cache pool will not give untrusted processes any more privileges, but just have the effect of confusing or breaking them

VI. Usability

A. Shadowing Policy

As untrusted processes run, they need to write into various files. On Windows, low integrity processes can only write into `%USERPROFILE%\AppData\LocalLow` for files and `HKEY_CURRENT_USER\Software\AppDataLow` for registry. Low integrity processes cannot write into other locations because these locations have medium or high integrity labels. Applications have to be rewritten so that they are mindful about where they can write.

SPIF transparently shadows modifications to benign files into a shadow directory. The shadow directory maintains the root directory hierarchy and provides a one-to-one mapping between benign and untrusted files. Untrusted files will be shadowed only if there exists a benign copy. When untrusted processes modify benign files, a copy of the file will be created in the shadow directory. Untrusted processes can then modify the untrusted copy instead. If untrusted processes attempt to create a file which does not exist, the file will be stored in the

user directory rather than being shadowed. Whenever untrusted processes access files, they will first check if shadow copies exist. If so, shadow copies will be accessed. Otherwise, the actual copies will be accessed.

While this policy converts all write denials into successful operations automatically, it is not always desirable. Shadowing provides an illusion that all write operations can be performed successfully on the original files, while some of the operations are actually diverted to another files. This diversion can lead to inconsistency in files. It is undesirable for users to edit a document while changes are redirected to another file. Hence, shadowing policy is only applied to non-data files. In our implementation, we redirect writes to `%USERPROFILE%\AppData` and `HKEY_CURRENT_USER`. On Windows XP, file system structures are less organized and we redirect all writes to files inside hidden directories located in user profiles. As applications are not expected to store user data in these directories, user data will not be shadowed.

B. User-Intention Driven Policy

SPIF permits transitions from normal to untrusted context only at the point of executing a program. A running process cannot change this context. This means that if a benign program is started normally, and it subsequently tries to open an untrusted file, this operation will be denied. However, it is possible that the only reason why the program was run was to open an untrusted file. Unless we infer this in advance and downgrade the process at the point of invocation, this use case cannot be supported.

To alleviate this problem, we seek to user user intentions when possible, and automatically apply appropriate policies to achieve what users want. In some ways, what we seek is similar to how Windows Protected Mode works currently: when users try to open a document from Internet, the application automatically starts itself in protected mode. However, this requires the application to be capable of running with different security levels, and consciously switch to the right mode for each input file. This approach provides the maximum flexibility, and can be supported in SPIF as well. An alternative is to infer intended use of *without any need for application modifications.* We describe our approach to accomplish this below.

The idea is to look at files that the applications open. If the files opened by the applications are likely to be data files (e.g., located in Desktop, Documents or Downloads directories) instructed by users, SPIF can treat them as user intentions to open the files. If subsequently the applications failed to open these files because these files are untrusted, SPIF can automatically spawn untrusted instances of the applications and open the required files. The explicitly accessed file idea in [17] can be applied in SPIF to further enhance the precision of the inference, by noting files explicitly specified by the users.

Note that all of this is applicable only only for benign applications. For untrusted applications, there is no way to distinguish user intentions because the applications may or may not be performing operations with user's consent or intent.

Document Readers	Adobe Reader, MuPDF
Editor/ Document Processor	MS Office, OpenOffice, Kingsoft Office, Notepad 2, Notepad++, CppCheck, gVim, AklelPad, IniTranslator, KompoZer
Internet	Internet Explorer, Firefox, Chrome, Calavera UpLoader, CCProxy, Tor + Tor Browser, Thunderbird
Media	Photoshop CC, Google Picasa, GIMP, WinAmp, Total Video Player, VLC, Picasa, Light Alloy, Windows Media Player, SMPlayer, QuickTime
Other	Virtual Magnifying Class, Database Browser, Google Earth, Celestia, Skype

Fig. 3. Software tested

VII. Evaluation

A. Software supported

As our system requires no modifications to applications, a wide range of applications can be supported naturally. These applications can run not only as benign processes as logged user, but also run as untrusted processes as shadow users. We tested the functionality of the applications listed in Figure 3.

Exceptions for Reading Untrusted files.

As benign processes run as R, compromising benign processes could compromise the integrity of the real users. Hence, the system enforces *no-read-down* [5] policy to protect benign processes from getting compromised. Denying reading files writable by untrusted users can sometimes break program functionality.

We consider certain objects as benign for reading. These include audio devices and pipes for communicating with various Windows services. Apart from these objects that are generic across all applications, some applications require additional configurations:

When we run Photoshop CC as benign process, it failed to start properly and suggested reinstalling the application. We found that Photoshop CC tries to access files in `C:\Program Files\Common Files\Adobe\SLCache` and `C:\ProgramData\Adobe\SLStore`. Accesses to reading files in these directories were denied by SPIF because they are world writable. After realizing it is the developer's intention to leave these directories and files as world-writable, we granted exceptions for Photoshop CC to read these files. After that, Photoshop CC can be started properly. (Note that this relaxation does not negate security guarantees — opens of untrusted files by benign Photoshop processes will still be recognized and stopped.)

While testing SPIF, we found that some files in the system directories are world writable. An example is `C:\Windows\System32\D3D9.dll`. When we used Windows Media Player to play a video, the video played without audio because accesses to the dll was denied. We found that the dll is actually digitally signed and we decided to add it to exception list. Once the dll can be loaded, Windows Media Player can then play the video.

File Size	500B to 5KB			5KB to 300KB			300KB to 3MB		
Operations	Unprotected	Benign	Untrusted	Unprotected	Benign	Untrusted	Unprotected	Benign	Untrusted
Files Created per Second	351.14	-5.02%	-10.45%	68.00	-2.79%	-2.02%	8.00	-1.25%	-1.56%
File Read per Second	350.14	-5.18%	-10.59%	67.64	-3.02%	-2.34%	7.60	-3.95%	1.97%
File Appended per Second	344.79	-5.19%	-10.58%	67.64	-3.02%	-2.61%	8.00	-2.50%	-2.34%
File Deleted per Second	350.21	-5.17%	-10.57%	67.86	-3.03%	-2.00%	8.00	-1.25%	-2.34%
Total Transaction Time (s)	285.36	6.53%	12.38%	367.29	3.05%	4.58%	308.67	1.27%	-0.62%

TABLE II
POSTMARK OVERHEAD

The rest of the processes can start as both benign and untrusted without any issues.

Reading both Benign and Untrusted files.

Applications like document readers and media players only read files. They do not edit data files. This allows untrusted document readers and media players to read both benign and untrusted files. On the other hand, benign readers and players are not allowed to read untrusted files.

Editing both benign and untrusted files.

SPIF does not allow a process to edit both benign files and untrusted files, as this can compromise the integrity of the benign files. However, it still supports editing both benign and untrusted files simultaneously, but in different processes. Users can edit benign files in benign processes, and edit untrusted files in untrusted processes. As these processes are running as different users, different instances of the same application can be supported naturally. However, the problem is to determine whether users want to open the untrusted files.

When it is the users intent to open untrusted files, the system should open the files with untrusted processes. However, when users do not expect opening the untrusted files, such an open should be denied. A question is how to determine user intent. We can consider user actions such as double clicking on the files, selecting files from a file dialog box, or explicitly typing the file names as indications of their intent. When intent is inferred in this manner, application is started in untrusted mode. Our system currently captures such intents via user interaction with the Windows Explorer: when users double clicked to open a file, Windows Explorer will execute the handler programs with the file path as argument. When the file path corresponds to an untrusted file, our system considers this as a user consent for starting a program in untrusted mode.

Untrusted processes modifying benign files.

TABLE III
SPEC2006 REF BENCHMARK

	Unprotected (s)	Benign (%)	Untrusted (%)
401.bzip2	1797	1.11%	0.79%
429.mcf	721	2.35%	-1.97%
433.milc	3383	0.32%	-0.67%
445.gobmk	1103	0.45%	-0.30%
450.soplex	1114	0.97%	-0.95%
456.hmmer	2398	0.52%	-0.27%
458.sjeng	1455	0.34%	-0.78%
471.omnetpp	760	1.14%	-0.42%
Average		2.75%	-0.26%

Applications like OpenOffice maintain runtime information in user profile directories. They expect these files to be both readable and writable, or they will simply fail to start. Having these files as benign would prevent untrusted processes from being usable. Letting these files become untrusted would break usability of benign processes.

SPIF shadows accesses to these runtime data inside user profile directories, hence benign and untrusted processes can both run without significant usability issues.

B. Performance

We compare the performance of the system under 3 scenarios:

- When there is no SPIF protection,
- When SPIF protection is enabled and the benchmark applications are run as benign processes, and
- When SPIF protection is enabled, and the benchmarks are run as untrusted processes.

We first benchmarked the system using SPEC2006, a CPU intensive benchmark. Table III shows the SPIF overhead on SPEC2006. CPU intensive benchmarks show negligible overhead because SPIF hooks on files and registry operations only. It incurs negligible overheads when for programs that make scant use of these API calls.

We also evaluated the system with postmark, a file I/O intensive benchmark. To better evaluate the system for Windows environment, we tuned the parameters to model files in a Windows 8.1 system. There were 193475 files on the system. The average file size is 299907 bytes, and the median is a much smaller 5632 bytes. We selected 3 size ranges based on this information: small (500 bytes to 5KB), medium (5KB to 300KB), and large (300KB to 3MB) bytes. Tests are run for multiple times and the average is presented. As expected, the system shows higher overhead for small files. This is because there are more frequent file create and delete operations. All these operations have to be translated into IPC to helper process to perform the operation on the user directory. As file size increases, the overhead becomes negligible.

We also benchmarked the system with Firefox. Specifically, we look at the time required to load webpages. We used the standard test suite to perform page load test. We fetched the top 1000 Alexa pages locally so that network variances are removed. Figure 4 shows the correlation between unprotected page load time with protected benign Firefox and untrusted Firefox. The overhead is -1.35% for benign Firefox and 0.5% for untrusted Firefox.

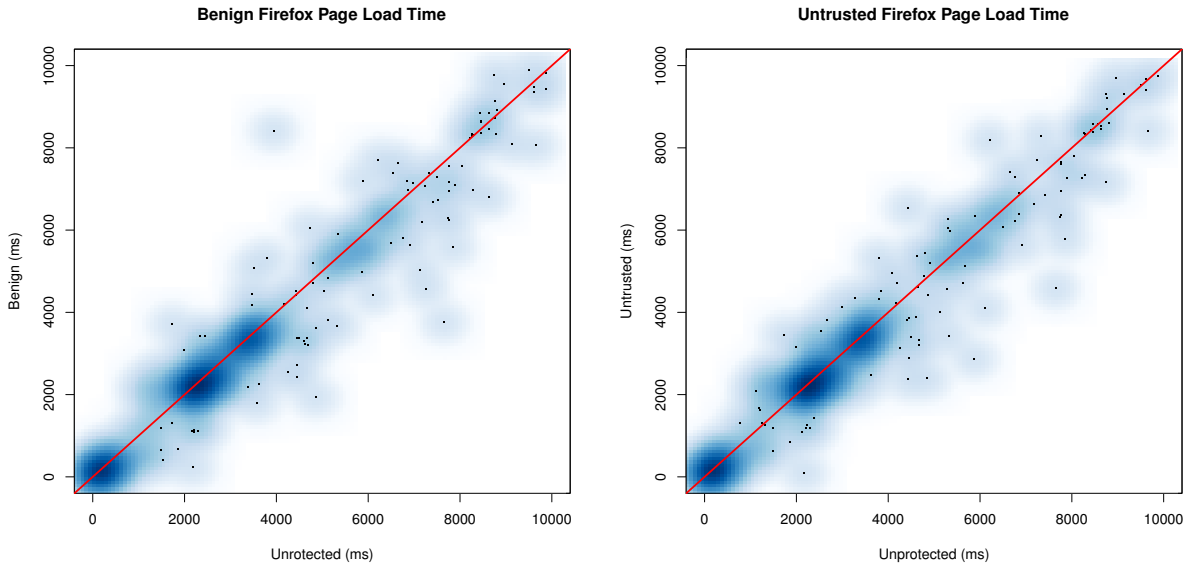

Fig. 4. Correlation between protected and unprotected page load time

CVE	OSVDB-ID	Application	Attack Vector
2013-4694	94740	WinAmp	Preference (ini)
2013-3934		Kingsoft Office Writer	Data (wps)
		Stuxnet PoC	Data (lnk)
	104141	Calavera UpLoader	Preference (dat)
2014-2013	102340	MuPDF	Data (xps)
	102205	CCProxy	Preference (ini)
	100619	Total Video Player	Preference (ini)
2013-6874	100346	Light Alloy	Data (m4u)
2014-0568		Adobe Reader	Code
2014-4114	113140	MS Windows	Data (ppsx)

Fig. 5. Malware tested

We note that many of the overheads are low enough that statistical noise begins dominate. This is the main reason why some of the overheads are negative.

C. Defense against malware

We evaluated the security of SPIF against "malware" available from exploit-db [13]. These malware were modeled mostly after real-world malware through reverse engineering exploits in the wild. They use the same mechanics that real-world malware exploits, except with a customizable payload. Figure 5 summarizes the exploits CVE/OSVDB-ID, vulnerable applications, and the attack vector.

These exploits can be classified into three types:

- data input attacks
- preference/configuration files attacks, and
- code attacks.

Both data and preference/configuration file attacks concern inputs to benign programs. When programs fail to sanitize untrusted inputs, attackers can exploit the vulnerabilities and take control of the applications. Data input attacks involve day-to-day files like documents (wps, ppsx, xps). They can be exploited by simply tricking users to open the files. On the other hand, preference/configuration files attacks exploit vulnerabilities in parsing configuration/preference files. Since these files are typically hidden from users, they are trickier to exploit directly. Instead they are often used together with code attacks to carry out multi-steps attacks to circumvent sandboxes.

This brings us to code attacks. Code attacks can exploit vulnerabilities in kernel or applications. SPIF does not deal with kernel exploits, where any users can simply invoke APIs with malformed parameters to escalate privileges. SPIF relies on the kernel to enforce policies. Hence, we focus on code exploits against applications. In this attack, attackers are able to execute code but with limited privileges. The goal is to escalate privileges. In the Adobe Reader exploit, it is assumed attackers have already compromised the sandboxed worker process. This exploit then circumvents the sandbox environment. Instead of running code outside of the sandbox directly, this exploit allows files to be created at arbitrary locations. It can drop a data file on user's desktop to carry out data input attacks. Or it can drop a malformed preference/configuration files to compromise benign applications when they run.

SPIF protects benign processes against all these attacks. Specifically, benign processes running with R privileges cannot open untrusted files. When users open a malicious data input, an untrusted process will run. Instead of compromising a benign process, attackers can only compromise an untrusted process running with U_R. In other words, the attack is unable to escalate privilege from "untrusted" to a real (i.e., benign) user. As a result, the system integrity is preserved.

As for the preference/configuration file attacks, malicious files cannot overwrite existing preference/configuration files. This is enforced by the OS via DAC permission. They can only be dropped in the shadowed location that is not accessible by benign processes. It is possible that these files will not be shadowed, but placed in the usual directory. This can happen when no benign preference/configuration files exist. However, as in the data attacks, SPIF protects benign processes from consuming these untrusted preference/configuration files, so this does lead to privilege escalation either.

While SPIF does not prevent code attacks from happening, its provenance tracking ensures that any effects from the code attacks are tracked. In the sandbox escape vulnerability, SPIF does not prevent dropping files outside of the sandbox environment. However, since both the Adobe Reader worker and broker processes are running as untrusted, any files created by the broker process are also automatically labeled as untrusted. This prevents the dropped files from compromising benign processes.

We specifically tested the Microsoft Windows OLE Package Manager Code Execution vulnerability, called Sandworm, on Windows 7. It was exploited in the wild in Oct 2014. When users view a malicious power point file, OLE package manager can be exploited remotely to run arbitrary code. It involves modifying registry in `HKLM` to run the payload. SPIF protects the system by running power point as untrusted users, which does not have permissions to modify the system registry.

VIII. Related Work

Information flow systems label subjects and objects so that policies can be enforced accordingly in a system-wide manner. The earliest works in this area date back to the 70's, with Biba integrity model [5]. Several recent works [16], [12], [11], [17] have been focused on making information flow work on contemporary OSes, especially Linux. Whereas these techniques have focused on centralized information-flow control, where the labels are global to the system, several recent efforts have developed more flexible notions of information flow called the decentralized information flow (DIFC) [18], [6], [10], which allows any principal to create new labels.

A main question for implementing an information flow system is how to label subjects and objects. The labeling scheme has to be applicable to both subjects and objects, and should be tamper-resistant against any untrusted subjects. Existing information flow systems rely on either rewriting the entire OS from scratch [18], [6] or making nontrivial modifications to the OS kernel [16], [7], [11]. The former approach is particularly appropriate for supporting new and fine-grained information flow models. However, for approaches that stress application compatibility or malware defense, an approach that avoids kernel modifications is much more suitable. This is even more true for Windows because there are no mechanisms such as LSM hooks. The fact that it is a closed source operating system makes it even harder to understand the Windows kernel. As a result, there have been very little efforts on developing information flow techniques on Windows.

Of the works described about PPI [16] was also focused on system-wide integrity protection and malware defense. Unlike us, their approach relies on kernel modifications (implemented using LSM hooks) for label propagation as well as policy enforcement. While such an approach provides more flexibility and hence supports a wider range of policies, its downside is that it is difficult to port to other OSes.

Among previous works, PIP [17] is the work that is most closely related to ours. Like SPIF, PIP also repurposes multi-user protection for information-flow tracking, and relies on the creation of an untrusted user and helper processes to ensure security while preserving transparency. But its design, targeted at Unix, necessarily differs from SPIF that targets Windows. Moreover, SPIF's design provides a greater degree of portability across different OS versions, and a higher level of application compatibility, having been applied to a much larger range of complex, feature-rich applications. While PIP also sports low overheads (e.g., 4% on Firefox), SPIF achieves further overhead reductions (e.g., $< 1\%$ for Firefox). Finally, it is difficult to meaningfully evaluate malware defense on platforms where there is a dearth of malware, whereas we present a much more convincing evaluation against malware in this paper.

We can consider SPIF as an extension to Ostia [8] and protected mode's sandboxing architecture. Instead of requiring applications to be refactored into worker and broker, trusted and untrusted component, we consider the whole process as untrusted, and have the helper process to provide controlled resource access.

Both user-driven access control [15] and our user-intention-driven policy focus on capturing user intention. However, the focus of user-driven access control is on granting resource accesses to untrusted applications, and their focus is on reducing additional user effort for granting these accesses, whereas our goal is to eliminate additional interactions. Especially in the setting of desktop systems, users are tired of dialog boxes asking for permission, and hence we seek a solution that avoids any additional effort on the part of users.

IX. Conclusion

In this paper, we presented a new approach for system-wide provenance tracking that is designed for Windows. Unlike existing malware defenses, which are reactive in nature, SPIF is proactive, and hence works against unknown and especially stealthy malware. We described the design of SPIF, detailing its security features, and features designed to preserve usability of applications. Our experimental results show that SPIF imposes low performance overheads, almost negligible on many benchmarks. It works on many versions of Windows, and is compatible with a wide range of feature-rich software, including all popular browsers and Office software, media players, Acrobat, and so on. We evaluated it against several

malware samples from ExploitDB, and showed that it can stop a variety of highly stealthy malware.

We certainly don't claim at this point that our prototype is free of vulnerabilities, or that it can stand up to any targeted attacks. But we do believe that any such weaknesses are the result of limited resources expended so far on its implementation, and are not fundamental to its design. Hardening it to withstand targeted, real-world malware attacks will require substantial additional engineering work, but we do believe that SPIF represents a promising new direction for principled malware defense.

We will make SPIF available as open-source software by the time of this conference.

References

[1] Annual report pandalabs 2013 summary, http://press.pandasecurity.com/wp-content/uploads/2010/05/PandaLabs-Annual-Report_2013.pdf.

[2] Cve-2010-3338 windows escalate task scheduler xml privilege escalation — rapid7, http://www.rapid7.com/db/modules/exploit/windows/local/ms10_092_schelevator.

[3] Detours.

[4] A. Barth, C. Jackson, C. Reis, et al. The security architecture of the chromium browser.

[5] K. J. Biba. Integrity Considerations for Secure Computer Systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[6] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*, 2005.

[7] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *S&P*, 2000.

[8] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[9] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

[10] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *SOSP*, 2007.

[11] N. Li, Z. Mao, and H. Chen. Usable Mandatory Integrity Protection for Operating Systems . In *S&P*, 2007.

[12] Z. Mao, N. Li, H. Chen, and X. Jiang. Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *TISSEC 14(3)*, 2011.

[13] Offensive Security. Exploits Database.

[14] PandaLabs. Quarterly report q2 2014 pandalabs, http://mediacenter.pandasecurity.com/mediacenter/wp-content/uploads/2014/07/Quaterly-PandaLabs-Report_Q1.pdf?_ga=1.34643817.933024576.1415762456.

[15] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 224–238, Washington, DC, USA, 2012. IEEE Computer Society.

[16] W. Sun, R. Sekar, G. Poothia, and T. Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P*, 2008.

[17] W. K. Sze and R. Sekar. A Portable User-Level Approach for System-wide Integrity Protection. In *ACSAC*, 2013.

[18] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI*, 2006.