7-2006

Ad hoc Extensibility and Access Control

Úlfar Erlingsson

John P. MacCormick
Dickinson College

Ad hoc Extensibility and Access Control

Úlfar Erlingsson
Microsoft Research
Silicon Valley
ulfar@microsoft.com

John MacCormick
Microsoft Research
Silicon Valley
jmacc@microsoft.com

Abstract

General-purpose, commercial software platforms are increasingly used as system building blocks, even for dependable systems. One reason for their generality, usefulness, and popular adoption is that these software platforms can evolve through *ad hoc extensions*: behavior tweaks outside the scope of supported platform interfaces. Unfortunately, such use of internal platform implementation details is fundamentally incompatible with security and reliability. Even so, platforms that exclude ad hoc extensions—for instance, by enforcing full isolation and strict interfaces—will, most likely, either have their security enforcement circumvented or be relegated to a niche market. In this paper, we identify ad hoc extensions as well as the economic and technical factors surrounding their existence. Subsequently, we propose the enforcement of novel access-control policies for reconciling ad hoc extensibility with security and reliability.

1. Introduction

Most computing today involves one or more general-purpose *software platforms* such as operating systems, databases, web browsers, or web servers. The vast majority of this activity occurs on only a select set of dominant, platform systems—the Windows and Linux operating systems [41, 43], the IIS and Apache web servers [25, 46], and the Oracle database system [27] are a few examples.

The ubiquitous use of these platforms raises natural security and reliability concerns: they comprise millions of lines of code and are not designed to guarantee behavioral properties. On the other hand, by using a well-established platform, software creators can leverage powerful advantages such as rich features, familiarity, standardization, and an implementation completeness otherwise hard to attain [24]. Therefore, a new software system is increasingly likely to be built on top of one of the dominant platforms—despite their lack of security, and empirical instability.

Extensibility is key to this success of software platforms: a platform's functionality can be extended, modified, or utilized through additional software [24]. These additions, or *extensions*, all act to customize the platform to a given application or purpose—whether they are scripts, binary machine-code modules, or even source-code patches. Unfortunately, support of extensions can have negative consequences; in particular, the interaction between an extension and its platform, or between extensions, can lead to security vulnerabilities or general instability that reduces the usefulness of the system. As a result, there is a long tradition of research that aims to make extensions more dependable by isolating them in memory, limiting their interactions to a few well-defined interfaces [4, 10, 23, 40], and by using cryptographic or other means to establish their trustworthiness [15, 31, 34]. Recently, there has been a resurgence of research interest in the topic of dependable, extensible systems [11, 19, 48, 56]; the motivation for this current paper has its origin in our own work on this topic [12].

Clearly, the instability and security vulnerabilities of the software platforms we use every day are very unsatisfying, and the authors of this paper are sympathetic to calls for the increased use of restricted interfaces. Yet, software often arises in unanticipated ways, and many key aspects of today's software platforms started out as only unsupported, expedient "hacks." Such *ad hoc extensions* provide important benefits to both the users and creators of a software system, and play a key role in its evolution into a dominant, general-purpose platform.

In this paper we argue that there is a substantial danger of being too restrictive when designing reliable, secure extensibility mechanisms. While it may ensure reliability—and be a necessary foundation for security—a design that isolates and restricts all extensions will forego some of the benefits of the "extensibility ecosystem". Specifically, such a system may not be amenable to the development of some features and, thus, may run the risk of reduced adoption.

This paper makes three contributions: (i) a definition of ad hoc extensions; (ii) an analysis of ad hoc extensions from the point of view of software development and industry economics; and (iii) a description of how to increase dependability without restricting extensibility by applying a novel form of access control to these ad hoc extensions. Although the topic of the paper is not amenable to absolute scientific rigor—and we must base our discussion on historical evidence and simple economic analysis—we believe our conclusions to be valid, and to have important consequences.

The remainder of the paper is as follows. In Section 2, we define ad hoc extensions, and give examples of their importance in achieving and maintaining the success of a software platform. Then, in Section 3 we analyze what gives rise to ad hoc extensibility, and whether it can be compatible with high-assurance dependability. Finally, in Section 4 we describe how to reconcile dependability with ad hoc extensibility through the enforcement of novel access-control policies.

We propose an architecture for such access control that ties support for pervasive mediation with both online management and behavior manifests for platform extensions. Section 5 offers our conclusions.

2. Identifying Ad hoc Extensions

Each extensible software platform, by its nature, exposes a set of software interfaces that support the programming of extensions. These public *extensibility interfaces* must be fully documented and provide all the functionality necessary for creating useful extensions. When an extension relies only on these documented, public extensibility interfaces, it can be said to be *well-behaved*.

To prevent undesirable behavior, public extensibility interfaces are deliberately of limited scope. (If those public interfaces are improperly designed—e.g., if they are too permissive—then well-behaved extensions may still cause undesirable behavior.) A software platform may, for example, allow extensions to append messages to a work-item queue that is naturally sorted by append order. By only exposing a single, limited append interface for this queue, the platform can hold this data structure abstract, maintain its ordering invariant, synchronize its access, and allow its implementation to change across platform versions.

Ad hoc extensions are not well behaved and do not restrict themselves to public interfaces. This characteristic, simply depicted in Figure 1, can serve to define them:

Definition: *An ad hoc extension depends for its functionality upon the implementation details of its underlying software platform—not only on the public extensibility interfaces specified and exposed by that platform.*

Because they circumvent the proper interfaces and rely on unsupported, undocumented platform details, there are many ways in which ad hoc extensions can cause undesirable system behavior. For instance, an ad hoc extension might accidentally corrupt the work-item queue, mentioned above, by not acquiring the correct lock before modifying its internal state.

Some ad hoc extensions are purely *passive*, and therefore less likely to violate system invariants. These extensions only observe, but do not modify, internal platform details, e.g., they may read undocumented fields in data structures. Screen scraping is one well-known implementation approach that fits this description. In the context of the work-item queue example, passive ad hoc extensibility could be used to provide a notification whenever messages are appended—something that is not possible given the limited append-only interface described earlier.

Other ad hoc extensions are *active*, and directly affect state in their host platform that should be opaque to them, e.g., they may modify an undocumented field in a configuration database. In particular, such active extensions can intercept and modify (in an arbitrary way) the arguments and return values of internal platform functions. In the example of the work-item queue, an active ad hoc extension would, e.g., be able to insert its own messages at the head of the queue by modifying its internal data structure. Note that—because of the limited public queue interface—such a modification may be necessary if the extension truly requires priority for its messages. On the other hand, this modification will violate the platform's invariant about the ordering of messages in the queue.

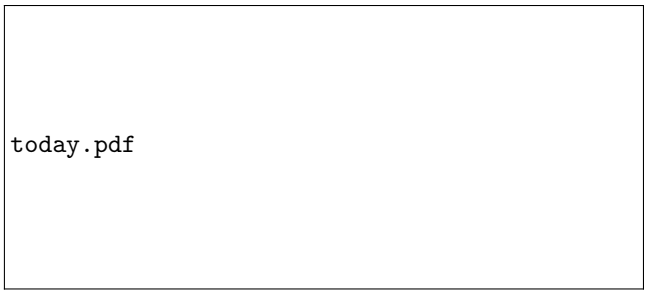

Figure 1. *Well-behaved extensions* interact with their platform via a public interface, whereas *ad hoc extensions* directly depend upon or manipulate the implementation details of their platform.

The above definitions notwithstanding, the concept of ad hoc extensions is best understood with the help of additional, real-world examples such as the ones below.

2.1 Examples of Ad hoc Extensions

This section briefly describes several examples of ad hoc extensibility, while an appendix gives the full details of one particular ad hoc extension. As with the work-item queue example described above, there is a fundamental reason why these examples rely on unsupported platform internals: the platform simply does not support the functionality required to implement them as well-behaved extensions. For instance, the platform may expose interfaces to query a system dictionary that is implemented as a hash table—but not provide any means for enumerating all of its keys. As a result, saving and restoring the dictionary contents may only be possible through implementation-dependent means.

Because of its relatively limited functionality, MS-DOS was the target of many clever extensions; we consider only a few of the numerous examples here. Third-party MS-DOS extensions added support for increased memory, longer filenames, and the caching and compression of data in secondary storage. These were all ad hoc extensions: implementing this functionality required knowledge and assumptions about unspecified aspects of how MS-DOS managed memory, implemented directory tables, and made disk accesses, respectively [39]. Subsequently, in later versions of MS-DOS, all of the abovementioned extensions were added as supported features, e.g., as the DPMI, VFAT, SmartDrive, and DoubleDisk technologies [8].

Historically, internationalization, localization, and accessibility have all been added to popular software platforms through a variety of of ingenious methods. Here, necessity has been the mother of invention: initial platform versions have typically not provided any support for these small, but

important, market segments. This omission may be understandable, since platform creators must focus their limited resources. But, as a result, this functionality could only be achieved through unsupported means, like the direct modification of implementation-specific, undocumented system fonts and string tables, as well as various forms of screen scraping. Even on a mature platform like Windows, "foreign" spell checking and certain Asian languages have, until quite recently, primarily been supported through de-facto-standard, third-party ad hoc extensions.

The nature of systems software (for example, Symantec's Norton SystemWorks [49] and Veritas Enterprise Backup [50], to name just two), is to augment an underlying software platform with new functionality. Because this augmentation is often unanticipated, systems software must often be implemented as ad hoc extensions. For instance, Norton's undelete feature must somehow interpret the undocumented on-disk structures of the Windows' NTFS file system. Similarly, backup on Windows NT was (until relatively recently) somewhat of a black art. As another example, mounting files containing disk images is not fully supported in Windows XP [36], yet many utilities exist for doing this—by definition, they are ad hoc extensions.

Debuggers, tracers, and other instrumentation form a special case of systems software. For this software, it is particularly important to observe or modify the behavior of arbitrary interfaces, and to arbitrarily change state. Interactive debugging, in particular, regularly involves modifying critical platform state such as the values of registers and the layout of the stack. In the Appendix, one example of an ad hoc extension in this category is described in full detail. Despite its intrusive nature, such systems software cannot always be categorized with ad hoc extensions—after all, platforms typically expose powerful debugging interfaces by design. However, ad hoc extensions are often implemented by using this platform debugging support in unintended ways.

In some cases, systems software changes the behavior of its underlying platform's most fundamental abstractions. Two examples, both third-party products, are VMware's implementation of a hosted virtual machine [47], and the multi-user support added to Windows NT by Citrix terminal services [6]. This suggests that no platform extensibility support can ever be complete: even the abstractions of that support mechanism may have to be extended—something that by its nature may require an ad hoc extension.

That a platform's extensibility interfaces can never be complete helps explain why—even on mature platforms—novel functionality is often implemented through ad hoc extension. Pop-up blocking for web pages is an illustrative example, as it is a recent addition to the mature, highly-competitive web browser platforms. Abstractly, a pop-up blocker needs to do two things: (i) mediate on all attempts to open web-browser windows, and (ii) block all those attempts, except when originating from a network domain listed as safe. Unfortunately, when the need for pop-up blocking first became apparent, most web browsers did not provide extensibility interfaces for receiving notifications about "window open" attempts and their associated source URL. When the web browsers were designed, such interfaces may have been complex to implement or seen as unnecessary; even today, some unwanted pop-ups cannot be correctly identified. By using ad hoc extensions, on the other hand, it was easy for third parties to implement pop-up blocking functionality for many popular web browsers, in a timely and relatively complete manner.

To conclude this section's examples, the following disparate features have all been implemented using ad hoc extensibility: support for scroll wheels on computer mice, anti-virus and anti-spyware defenses, the desktop remoting of GoToMyPC [7] and WebEx [7], the Google toolbar for web browsers [16], file and email encryption, and Bluetooth networking. These ad hoc extensions all use different means to achieve their goals: for instance, graphical Windows software often uses code injection, window subclassing, and interface detouring [18, 57]. Such graphical software will depend on the names of particular window "classes" and the ordering of particular window "events", both of which are implementation specific.

The discussion so far has highlighted the benign, positive aspect of ad hoc extensions, where they extend the underlying platform with new, useful functionality. There are at least as many cases where ad hoc extensions have been used unnecessarily, for frivolous purposes, and with dire consequences to platform integrity. For example, on MS-DOS, utilities for performance optimizations such as early versions of SmartDrive, modified disk semantics to use write-back caching—with the unfortunate side-effect of possible file system corruption. Needless to say, utilities that did not immediately remedy this flaw never gained popularity.

Purposefully malicious ad hoc extensions include viruses, spyware, kernel rootkits and other malware. It is therefore not surprising that recipies for malware creation, e.g., the recent book on Windows kernel rootkits [17], are little more than an enumeration of ad hoc extensibility particulars.

Whatever their purpose, the use of unsupported interfaces makes all ad hoc extensions inherently fragile, unreliable, and intolerant of platform implementation modification—in particular, modifications due to other ad hoc extensions. On many popular platforms this has led to the development of entire frameworks for making ad hoc extensions more robust (e.g., by avoiding bad interactions between them), and for their general management and support. Examples of such frameworks include HackMaster for the Palm Pilot [9], Extension Overload for Apple's Mac OS 9 [32], and APE Application Enhancer for Mac OS X [51]. In Section 4 we describe mechanisms that offer further support.

3. Ad hoc Extensibility: Causes and Effects

As touched upon in the previous section, there are both positive and negative aspects to ad hoc extensibility. In this section, we look at ad hoc extensibility in further detail, and consider both what drives the creation of ad hoc extensions (e.g., time pressure), and what can result from their existence (e.g., reduced reliability).

3.1 The Software Development Process

There many ways that ad hoc extensions can arise during software development. Sometimes, the cause is a combination of expediency and the ambiguous nature of platform interface documentation. A programmer is unlikely to investigate fully what is the precisely correct use of interfaces; rather, she is likely to simply invoke them in the first manner she discovered to provided the proper functionality. If she applies good engineering practice, then she will also confirm with a suite of tests that the interfaces continue to behave as expected under a variety of inputs and environment conditions.

Unfortunately, platform interfaces may consistently provide certain, desirable functionality not by design, but as a mere side-effect of their implementation details; relying on such functionality can make any software implementation-dependent. The less careful the programmer is, the more likely is the creation of such *inadvertent ad hoc extensions*. As one example, the order of messages in the Windows graphical user interface has always been very ill-defined, although quite consistent on any given instance of the operating system. Therefore, it is difficult to create advanced, yet version-independent, graphical Windows applications. Not surprisingly, for the last ten years, versions of Windows have included an "application compatibility flags" database that changes the per-application order of these messages to make popular software run correctly [30].

Of course, developers will in some cases deliberately create ad hoc extensions, as shown by the many examples given previously in this paper. This may be done either to expedite the implementation of an extension, or to overcome limitations in public platform interfaces.

Whether they are inadvertent, or created on purpose, the mere existence of ad hoc extensions can severely hamper the creation of a new platform version. In the limit, this can result in *implementation ossification*, where almost no implementation detail can be modified, because the chances are that some software is relying on it. Some platforms, by design, expose more of their structure and they are therefore more likely to suffer such ossification. Elaborate, object-oriented class hierarchies seem particularly vulnerable to this effect; this is likely the reason why Sun's Java platform is currently supported in five parallel, similar versions.

3.2 Security, Reliability, and Assurance

No matter how it comes about, extensions' dependence on internal platform details can have negative effects on both security and reliability. Indeed, most viruses and other malware could be categorized as stealthy, ad hoc extensions.

It is easy to see that ad hoc extensibility is fundamentally incompatible with formally-verified, high-assurance dependability guarantees, e.g., for security and reliability. Because it may potentially do anything, an ad hoc extension can act to prevent progress (obstructing availability), read any data (breaching secrecy), and write any combination of values (violating integrity invariants).

Even when there is a proof, whether formal or informal, of these platform guarantees, they may still be invalidated: to be tractable, proofs must abstract from implementation details [22], but an ad hoc extension can make use of, or make changes to, intermediate platform states that are not captured in these proof abstractions. Such problems sometimes arise in real-world systems; for instance, when paging was added to the Tenex platform, exposing additional intermediate platform state, suddenly passwords could be trivially broken character-by-character [1].

For most practical purposes, however, ad hoc extensions can be compatible with reasonable levels of dependability. Through use of testing, inspection, and problem reporting, any violation of system invariants that is likely to occur is likely to be detected. Although such reviews often miss vulnerabilities, and may not detect many information-disclosure channels, they are the only process currently used to assess most commodity software. Clearly, this process can also be applied to ad hoc extensions, even when highest software standards must be met. Evidence to this effect is the universal use of anti-virus and virtual-machine software—both ad hoc extensions, typically—in the dependability-conscious realm of enterprise computing.

3.3 Market Forces for Software Platforms

Ad hoc extensions can add remarkable value to a software platform, as shown by the examples earlier in this paper. These benefits extend to all platform stakeholders, including its creators, its users, and its third-party developers, and can help in creating an "ecosystem" that propels the platform into a dominant market position.

In many cases, new versions of extensible platforms will directly support functionality previously only available through implementation-specific means. Such direct support is especially likely when several popular extensions have made repeated use of the same unsupported, clever hack. This delayed support of important features holds an advantage for the creators of general-purpose software platforms: any given platform version need not support all its possible uses, as long as new, interesting features can be implemented by ad hoc extensibility. The next version can incorporate support for whatever turns out to be important in the marketplace—e.g., as the recent "Tiger" version of Mac OS X did for the kernel interfaces [42]. The opposite approach, where all possible uses are supported from the beginning, requires platform creators to be almost omniscient, and can guarantee only a more complex design and a higher implementation cost for the platform.

There have been few serious attempts at providing platform interfaces that support all possible applications; invariably, some developers find the results too restrictive—as has happened, e.g., in the case of Linux Security Modules [28].

An inflexible platform interface is also frustrating, since it may not support features that its users consider of critical importance. Even when this support can be had through negotiations with the platform creators, the time and cost of doing so forms a major obstacle. It can be argued that the

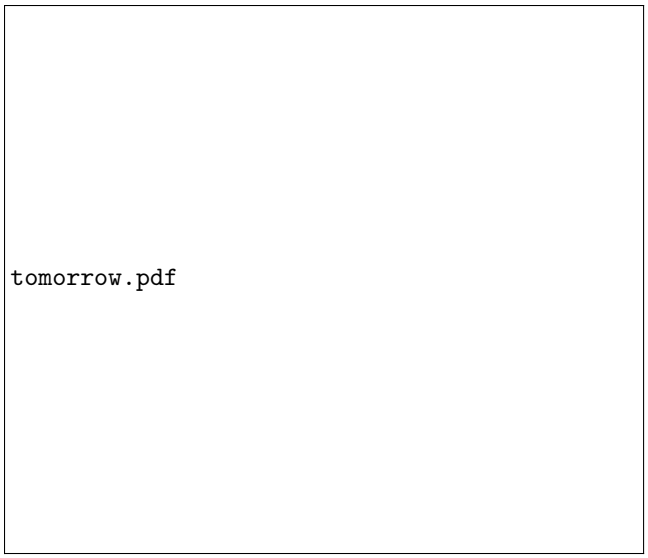

Figure 2. Our proposed architecture for making ad hoc extensions more dependable through the use of a mediation framework, the gathering and publication of online monitoring data, and behavior manifests that accompany extensions.

lack of such impediments is a key driving force behind the recent success of open-source software platforms.

General-purpose platforms with strong dependability guarantees, as intended in current research projects, like Asbestos, Mondrix, and Singularity [11, 19, 56], will hopefully become a reality someday—and this will involve restricting extensions and making them use only well-defined interfaces. But, without admitting the benefits of ad hoc extensions, market forces may result either in these platforms becoming closed, niche-purpose systems, or in the subversion of their reliability mechanisms.

4. Access Control for Ad hoc Extensions

We believe that future general-purpose platforms can reconcile ad hoc extensions with reliability and security and—in doing so—garner all the practical, economic advantages of ad hoc extensibility described in this paper. Specifically, we propose that platforms implement the following three-part architecture:

1. The platform provides a *mediation framework*: a mechanism that identifies platform implementation details and allows extensions to arbitrarily observe, modify, and interpose on them—but forces extension to openly declare if, and how, they use this framework.
2. The platform is part of an *online monitoring infrastructure* that identifies software, monitors behavior and its compliance with declared intent, and makes these statistics public.
3. All platform extensions come with a *behavior manifest*, written by the extension vendor, that statically declares how the extension intends to interact with the platform at runtime—describing, in particular, any use of the mediation framework.

The details of our proposed architecture are given in the next four subsections; Figure 2 depicts an outline of our proposal, which can be contrasted with Figure 1.

The goal, however, can be described here: in combination, the three ingredients of our proposed architecture should give popular extensions a strong incentive to be well-behaved, yet still leave room for outside-the-box originality. Furthermore, our proposal should serve to provide stakeholders with all the information they need to assess, regulate, and tolerate the use of ad hoc extensibility.

Of course, our proposals do not fully reconcile ad hoc extensibility with reliability and security; as argued in Section 3.2, this appears impossible. However, an implementation of our architecture seems likely to be often preferable to the alternatives: doing nothing—as is the current situation on dominant platforms—or enforcing strict isolation and conformance to interfaces, thereby forgoing all the benefits that accrue from ad hoc extensions.

Many of the issues of ad hoc extensibility are addressed in our proposed architecture through synergy between its three parts. For example, it might seem that the mediation framework would exacerbate the problem of implementation ossification (described in Section 3.1). However, manifests and use of the mediation framework, coupled with online reporting, would give platform creators—when changing internal implementation details—full knowledge about any existing dependencies on those details. This knowledge allows platform creators to discover what backward-compatibility support is needed, make changes contingent on the impact to popular, important software, notify vendors, and negotiate with them, as well as prevent incompatible software from executing on new platform versions. Thereby, our architecture greatly improves on the current situation, which is characterized by uncertainty and exemplified by Windows' manually maintained "application compatibility flags".

Implementations of our architecture could be driven by market forces—e.g., in a manner similar to today's antivirus and anti-spyware industries. The bulk of the work, such as operating the monitoring infrastructure, might not be performed by platform creators, or even administrators, but rather by third-party stakeholders such as commercial IT management firms, insurers, or even governments; such third parties might have the most to gain from increased security and reliability. Similarly, a differentiated market might apply to the creation and use of behavior manifests.

Even partial implementations of our architecture would have benefits. The online monitoring infrastructure, by itself, might be sufficient to permit less draconian administration of corporate or university computers, without decreasing reliability—e.g., users might be allowed to install all extensions with a proven reliability history. Alternatively, the mediation framework, described next, can serve to reduce instability, even when implemented in a simplified manner.

4.1 Exposing the Platform to Mediation

The mediation framework extends the idea of interposition [18, 20] to all internal platform interfaces and data struc-

tures. In this way, the mediation framework is closely related to program instrumentation [44, 45], language-level reflection [54], aspect-oriented programming [21], and inlined reference monitors [13].

To allow for their mediation, as many internal platform implementation details should be enumerated and named as are possible to identify. Those names can be created automatically by basing them on the original, programming-language identifiers used for code, variables, and data types. In fact, executable binaries compiled from modern languages (such as C# and Java) already include those names in order to support runtime reflection. In the case of hand-written assembly language, and legacy C code, the required names can be automatically extracted from compiler output such as the symbol files used by debuggers.

Platform creators already provide identifying information to extension writers; for instance, the debug symbols for any Windows binary can be freely retrieved from a public network service run by Microsoft [29]. As part of the mediation framework, this data would be augmented, e.g., to include certain details—such as the full names for internal, local variables—that are currently elided. These additions could provide metadata useful for assessing the impact of ad hoc extensions. The platform creators could, in particular, put a value on the risk of mediating certain components and further annotate internal data structures as to whether they are critical to security or reliability; in an operating system, the stored hash of the user's password and the hardware interrupt dispatch table might be expected to carry such an annotation.

4.2 Supporting Mediation by Ad hoc Extensions

In addition to naming, the mediation framework should provide a mechanism that enables—yet regulates—the use of platform implementation details by ad hoc extensions.

This mediation mechanism should give highly flexible access to platform internals; ideally, it should not only be possible to mediate every internal function invocation, and observe or modify arguments and local variables, but it should even be possible to mediate the reading or writing of certain data structures, given their named identifier or data type. A primary concern of this platform mechanism would be supporting the insertion of *mediation hooks*; this is sufficient, as long as ad hoc extensions can designate arbitrary code to be executed at the point of each such insert hook. Such a mechanism could be implemented in several ways, e.g., by dynamically detouring running code [18] or instrumenting new code as it is loaded [13, 44].

Because it allows for the access control of ad hoc extensions, this mediation mechanism lies at the heart of our proposal. This access control enforcement would proceed through decisions based on information in manifests, the actual runtime behavior of each extension, as well as a local policy, set by relevant stakeholders (e.g., administrators).

In particular, only an ad hoc extension that declared itself as such, in its manifest, would be able to use the mediation mechanism. The extension's identity, or that of its vendor, might be also grounds for rejection, e.g., based on either a purely local policy or on the published statistics about the extensions—such as whether its actual runtime behavior complies with its manifest. Furthermore, access control decisions could be made depending on a categorization of the ad hoc extensions, e.g., based on to what, and how many, implementation details they depend upon; thus, passive and active ad hoc extensions might be treated separately, as might one that mediates high-risk details.

In addition to enforcing policy, the mediation mechanism could perform additional access control in order to increase the reliability of the ad hoc extensions. This gatekeeping could eliminate unexpected, cross-extension interactions and other common causes of instability. For instance, the mediation mechanism should take no action unless the versions of all affected platform components conformed with the versions given in the ad hoc extension's manifest. The mediation mechanism might also choose to honor an extension's request for exclusive mediation on some internal details—although it might also interpret such requests as suggestive of high-risk behavior. Perhaps most importantly, the mediation mechanism could give synchronization guarantees to ad hoc extensions, both for their installation activity and for the execution of their mediation code. Such synchronization could range from a coarse-grained, single lock, to locks per component, per interface, or even per named identifiers or data types.

While the above type of access control can reduce the chances of reliability and security faults, it cannot eliminate them; in particular, it cannot provide any protection against malicious ad hoc extensions. However, our proposed access control can successfully limit ad hoc extensions to those that meet high standards—e.g., to only those that passive monitor non-critical platform components and have a proven reliability record. For some examples (such as that of the Appendix), the stronger version checking and synchronization support alone can succeed in eliminating almost all risks related to the extension.

4.3 Online Behavior Monitoring of Extensions

Aspects of an online monitoring infrastructure already exist: automated bug reporting and software updates are commonplace in many systems. However, more is required. First, the details and automation of bug reports must be increased, e.g., as described in [37]. Second, failure statistics must become public if administrators—or the stakeholders gaining most from increased dependability—are to make informed decisions based on software reliability records. And third, reports are needed about both healthy and problem-prone machines, as shown by systems such as Cooperative Bug Isolation [26], Strider [53], and PeerPressure [52], and proposed in [5].

Only with such detailed reports can the cause of an extension's stability problem be discovered and fixed, e.g., through negotiations between the extension writer and the platform creators. In addition, practical details must be addressed, such as how to assign meaningful identities to software, its creators, users, as well as administrators. Such

identities must allow for software versioning, updates, and patches, and might be based on existing schemes [15, 31].

Public dependability data will reduce the popularity of unreliable extensions and, thereby, encourage that any underlying problems be addressed. The reliability of a software extension, as reported by the online infrastructure, evolves in a natural fashion. Initially, when it is only used by a few early-adopters, the infrastructure would report the software's reliability as essentially unknown; however, with popularity, and pervasive coverage of its behavior, a more accurate reliability estimate would become available. Eventually, detailed assessments could be given, such as "with 95% confidence, using extensions X and Y together is causally linked to k additional reliability issues, per month".

As mentioned earlier, when used in our architecture, the online monitoring infrastructure would also help prevent implementation ossification. It allows the easy identification of which popular, important software needs help in switching to supported extensibility interfaces; it also allows platform creators to proactively work to prevent ad hoc extensibility from becoming entrenched in the first place. If online monitoring also reports on the use of supported extensibility interfaces, it can further support a platform's evolution by helping to regulate inadvertent ad hoc extensions.

4.4 Manifests for Circumscribing Behavior

There are many techniques for specifying and using behavior manifests, or contracts [2, 3, 14]. A manifest in our architecture is a statement attached to an extension in which the extension writer declares what platform interfaces the extension uses, and—in the case of ad hoc extensions—how they rely on platform implementation details using the mediation framework. It is left to the relevant stakeholders to choose what they see as acceptable manifests. Instead of being purely descriptive, these manifests can be what Lampson [24] calls "specs with teeth," i.e., either through runtime enforcement or static analysis, extensions can be guaranteed to comply with their manifests. It is important that any violations of the manifest, whether during admission testing or actual use, get reported and have negative consequences.

Using the existing online mechanisms for platform updates, the manifest of installed extensions could be updated over time, e.g., to include new version data, as vendors confirm that the extensions function well with new versions of platform components. (To allow uninterrupted use of extensions, the platform creators might especially facilitate such version data updates, e.g., by giving vendors ahead-of-time access to changed platform components.)

In this manner, the mediation framework facilitates, but regulates innovative, clever hacks. All ad hoc extensions will naturally be suspect and thus—all other things being equal—less popular, thereby giving extensions a strong incentive to being well-behaved. In particular, an ad hoc extension that mediates high-risk, critical platform internals will find few takers until it has established a track record as being reliable. However, for advanced, cutting-edge extensions, as well as single-use, custom extensions, such admissions in the manifests may well be considered as reasonable.

4.5 Implementing the Proposed Architecture

It is easy to see how our proposed architecture could be implemented for software platforms based on the Java language runtime. Java-based systems already support both code manifests and strong cryptographic code identity [15]; many already contact the platform creators through an online service for sending error reports and retrieving updates; and it has been demonstrated that it is simple to accommodate both the identification and hooking required for the mediation framework [13, 54]. Indeed, architectures similar to our proposal have already been implemented for Java-based application server platforms, the Wily Interscope management and monitoring system being one example [55].

There are several benefits to implementing our architecture on top of a type-safe platform, such as Java. For one, its access control could be made uncircumventable—i.e., the mediation framework, manifests, and online reporting could themselves be outside the realm of ad hoc extensibility [13]. Type safety can also help guarantee that extension's runtime behavior corresponds to their manifests. For instance, for an ad hoc extension that purports to be purely passive, the ad hoc extension code inserted by the mediation mechanism can be prevented from modifying any internal platform state [13]. (Such restrictions would, of course, make it impossible to implement some, potentially useful ad hoc extensions; for many purposes, this loss will be outweighed by the benefits of stronger guarantees.)

Of course, in their current form, the existing Java mechanisms are not sufficient for a realistic implementation of our architecture. Java code manifests would have to be augmented with declarations for specifying use of the mediation framework by an ad hoc extension. The mediation framework itself, and its access-control decision procedures, would have to be created (including any required instrumentation mechanisms, e.g., based on the Java ClassLoader, as in [13]). Finally, the rudimentary online servicing of existing systems would have to be tied into a full-fledged online monitoring infrastructure that can provide public dependability data. However—although it would require a significant effort, both in terms of software development and in terms of standardization and consensus-building between the relevant stakeholders—there are no technical obstacles to an implementation of our proposed architecture for Java-based platforms.

5. Conclusion

Ad hoc extensions provide many benefits: they enable developers to create truly innovative features, they allow users to benefit early from functionality, and they enhance the value of software platforms in ways that help them achieve and maintain dominance. Therefore, even though kludgy hacks are an enemy of dependability, in practice their omission might prevent the success of future secure, reliable platforms. The controlled support of arbitrary tweaks, e.g.,

through our proposed architecture, can help reconcile dependability and ad hoc extensibility.

Acknowledgments

Several of our colleagues at Microsoft Research, Silicon Valley, and Microsoft Research, Cambridge, provided helpful feedback on earlier versions of this paper, Comments from Tim Harris, Doug Terry, and Lidong Zhou were particularly useful.

References

[1] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[2] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system. In *Proc. CASSIS'04*, 2004.

[3] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *J. Syst. Softw.*, 65(3), 2003.

[4] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. SOSP'95*, 1995.

[5] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer. Using runtime paths for macroanalysis. In *Proc. HotOS'03*, 2003.

[6] Citrix Systems, Inc. Citrix Terminal Services. `http://www.citrix.com/`.

[7] Citrix Systems, Inc. GoToMyPC. `http://gotomypc.com/`.

[8] J. Cooper. *Special Edition Using MS-DOS 6.22*. Que, 3 edition, 2001.

[9] DaggerWare. HackMaster 0.9 for the original Palm Pilot. `http://www.palmblvd.com/software/pc/HackMaster-1999-02-21-palm-pc.html`.

[10] R. Daley and J. Dennis. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM*, 11(5), 1968.

[11] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, and M. K. R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. SOSP'05*, 2005.

[12] Ú. Erlingsson, T. Roeder, and T. Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-2005-82, Microsoft Research, 2005.

[13] Ú. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *Proc. of 2000 IEEE Symposium on Security and Privacy*, May 2000.

[14] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI'02*, 2002.

[15] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[16] Google, Inc. Google Toolbar. `http://toolbar.google.com/`.

[17] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[18] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. of the 3rd USENIX Windows NT Symposium*, 1999.

[19] G. Hunt and J. Larus. Singularity design motivation. Technical Report MSR-TR-2004-105, Microsoft Research, Dec. 2004.

[20] M. Jones. Interposition agents: transparently interposing user code at the system interface. In *Proc. SOSP'93*, 1993.

[21] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *Proc. OOPSLA '03*, 2003.

[22] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, 2002.

[23] B. Lampson. Protection. In *Proc. 5th Princeton Conf. Information Sciences and Systems*, 1971. Reprinted in ACM Op. Sys. Rev. 8, 1 (Jan. 1974), pages 18-24.

[24] B. Lampson. Software components: Only the giants survive. In *Computer Systems: Theory, Technology, and Applications: A Tribute to Roger Needham*. Springer, 2004.

[25] B. Laurie and P. Laurie. *Apache: The Definitive Guide*. O'Reilly & Associates, 3 edition, 2002.

[26] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Public deployment of cooperative bug isolation. In *Proc. RAMSS'04*, 2004.

[27] K. Loney. *Oracle Database 10g: The Complete Reference*. McGraw-Hill Osborne Media, 2004.

[28] J. P. Mello, Jr. Developer raps Linux security. *LinuxInsider*, January 11th, 2005.

[29] Microsoft Corp. Debugging tools and symbols. `http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx`.

[30] Microsoft Corp. Windows application compatibility. `http://msdn.microsoft.com/compatibility/`.

[31] Microsoft Corp. Frequently asked questions about Authenticode, 2000. `http://msdn.microsoft.com/library/en-us/dnauth/html/signfaq.asp`.

[32] T. Ming and S. Mitchell. Extension Overload. `http://www.xoverload.com/extensionoverload/`.

[33] G. Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, 2000.

[34] G. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[35] W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, 2 edition, 2002.

[36] Open Systems Resources, Inc. Peter pontificates. *The NT Insider*, 11(4), Dec 2004. `http://www.osronline.com/`.

[37] J. Redstone, M. Swift, and B. Bershad. Using computers to diagnose computer problems. In *Proc. HotOS'03*, 2003.

[38] M. Russinovich and B. Cogswell. SysInternals. `http://www.sysinternals.com/`.

[39] A. Schulman. *Undocumented DOS: A programmer's guide to reserved MS-DOS functions and data structures*. Addison-Wesley, 1990.

[40] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. OSDI'96*, Oct. 1996.

[41] E. Siever, A. Weber, and S. Figgins. *Linux in a Nutshell*. O'Reilly & Associates, 4 edition, 2003.

[42] J. Siracusa. Mac OS X 10.4 Tiger: Kernel updates. *Ars Technica*, page 4, April 28th, 2005. `http://arstechnica.com/reviews/os/macosx-10.4.ars/4`.

[43] D. Solomon and M. Russinovich. *Windows Internals*. Microsoft Press, 4 edition, 2005.

[44] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[45] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report WRL Research Report 94/2, Digital Equipment Corporation, 1994.

[46] W. Stanek. *Microsoft IIS 6.0 Administrator's Pocket Consultant*. Microsoft Press, 2003.
[47] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proc. USENIX'02*, June 2001.
[48] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Computer Systems*, 22(4), Nov 2004.
[49] Symantec Corp. Symantec Norton SystemWorks 2005. `http://www.symantec.com/sabu/sysworks/basic/`.
[50] Symantec Corp. Veritas Backup Exec. `http://www.veritas.com/`.
[51] Unsanity LLC. APE Application Enhancer. `http://www.unsanity.com/haxies/ape/`.
[52] H. Wang, J. Platt, Y. Chen, R. Zhang, and Y. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proc. OSDI'04*, 2004.
[53] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. Wang, C. Yuan, and Z. Zhang. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proc. LISA*, 2003.
[54] I. Welch and R. Stroud. Kava - A reflective Java based on bytecode rewriting. In *Proc. 1st OOPSLA Workshop on Reflection and Software Engineering*, 1999.
[55] Wily Technology, Inc. Interscope. `http://www.wilytech.com/solutions/products/Introscope.html`.
[56] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proc. SOSP'05*, 2005. `http://www.cag.lcs.mit.edu/scale/papers/mmp-sosp2005.pdf`.
[57] F. Yuan. *Windows Graphics Programming: Win32 GDI and DirectDraw*. Prentice Hall, 2001.

Appendix: The Details of the TDImon Ad hoc Extension for Network Monitoring

The SysInternals website [38] is a source of many useful Windows systems software utilities created by Mark Russinovich, a co-author of the definitive book on the internals of the Windows operating system [43]. Most of these utilities rely on some unsupported properties, e.g., they often make direct use of the undocumented NT system call interface [33]; therefore, they are ad hoc extensions to Windows. (Interestingly, techniques for Windows kernel rootkits are frequently based on the available source code for these utilities [17].)

TDImon is one of these utilities. It monitors activity at the kernel's Transport Driver Interface (TDI), which is fundamental to Windows' implementation of many network protocols, including TCP/IP [35, 43]. TDImon comprises a user-mode application and a kernel-mode driver.[1] The application displays a listing of network operations—such as accept, connect, and send—and the user process, IP addresses, etc., associated with each of those operations. This information is gathered by the driver, which the application dynamically loads into the Windows kernel.

It is the kernel-mode portion of TDImon that relies on unsupported, undocumented implementation details of the Windows TCP/IP TDI driver. To be precise, TDImon overwrites a table of seven function pointers in an internal, undocumented data structure of the Tcpip TDI driver instance, thereby changing what functions are invoked for TCP/IP network operations. Once overwritten, this table points to seven functions implemented by the kernel-mode TDImon driver; each of these TDImon functions records information about the network operation and then invokes the original function pointer for that operation.

TDImon gains several benefits from its use of ad hoc extensibility. In particular, it can simultaneously achieve three goals: (i) the utility can be used immediately after installation, without an intervening operating system reboot; (ii) it can identify the correct user-level process responsible for each network operation; (iii) it can be run alongside other network filters (such as for firewalls or VPNs). In Windows, it is difficult for well-behaved extensions to achieve these goals for two reasons: first, the set of network filters is limited, and mostly statically determined at boot time; second, data about user-mode processes is not always propagated to those filters. In the past, some Windows networking software has gone to great lengths to achieve these goals through alternate means, e.g., replacing the Windows' system-wide user-mode networking library (the WinSock DLL), and dynamically detouring running applications [18]. Compared to this, changing the TDI function-pointer table is a simple and reliable way to achieve these goals.[2]

At the same time, TDImon runs several risks by its unsupported behavior. For instance, since it is undocumented, nothing guarantees that the TDI function-pointer table will always be used in the manner TDImon expects. More worryingly, another ad hoc extension, or some unexamined platform component, might be critically dependent on the values of that table. These risks are mitigated by TDImon's purely passive nature, once installed, and by its use of defensive programming, such as careful version checking and atomic overwriting of pointers. The remaining risk may be considered acceptable, given TDImon's past and present popularity and the lack of reported problems. Of course, through the use of a mechanism for supporting reliable ad hoc extensibility, e.g., based on our architecture from Section 4, this risk could be virtually eliminated.

[1] In Windows, the term "driver" is used for for device drivers, loadable kernel modules, and even for parts of the operating system itself.

[2] The enhanced platform support of the upcoming Windows Vista makes it easy to achieve these goals—and dynamically add filters like TDImon—as the discussion of Section 3.3 would predict.