Simple and Practical Integrity Models for Binaries and Files

Yongzheng Wu, Roland Yap

Simple and Practical Integrity Models for Binaries and Files *

Yongzheng Wu[1] and Roland H.C. Yap[2]

[1] Huawei
Wu.Yongzheng@huawei.com
[2] National Univ. of Singapore
ryap@comp.nus.edu.sg

Abstract. Software environments typically depend on implicit sharing of binaries where binaries are created, loaded/executed and updated dynamically which we call the *binary lifecycle*. Windows is one example where many attacks exploit vulnerabilities in the binary lifecycle of software. In this paper, we propose a family of binary integrity models with a simple and easy to use trust model, to help protect against such attacks. We implement a prototype in Windows which protects against a variety of common binary attacks. Our models are easy to use while maintaining existing software compatibility, i.e. work with the implicit binary lifecycle requirements of the software and assumptions on binary sharing. We also propose a conservative extension to protect critical non-binary files.

1 Introduction

It is typical in software environments that the software consists of a collection of software components in the form binaries such as executables, dynamically linked libraries (DLLs), plugins, drivers, etc., e.g. this is the case in Windows. Binaries may be shared and used (executed/loaded) by many software, e.g. Windows Office software components are shared by programs in the Office suite. Binaries are usually created when a software is installed. Software updates modify/delete existing binaries or create new ones. Software uninstall usually deletes binaries. We call the creation, usage, sharing, modification and deletion of binaries associated with software, the *lifecycle of binaries*.

Binaries often have a complex and dynamic lifecycle with many kinds of interactions (arising from functionality, usability and software development reasons). However, the binary lifecycle is also exploited in attacks, e.g. a Java malware (EUR:Backdoor.Java.Agent.a [1]) exploits a vulnerability (CVE-2013-2465) to copy itself to the user home directory and launch on system startup. This attack

* This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

shows exploitation of the binary lifecycle in two ways: (a) it uses operating system mechanisms which can load or execute binaries; and (b) the malware uses binaries so that it becomes persistent.

Windows is a primary target for binary attacks. It has mostly implicit sharing of binaries and co-dependencies between binaries, e.g. Firefox uses other software plugins, Windows Explorer uses third party codecs, etc. Windows also has a large attack surface [12, 14] with many mechanisms for running executables or loading binaries which is used both in the binary lifecycle but also exploited in attacks.

Many security models and mechanisms [10, 11, 13, 15, 16, 21] have been proposed to protect against binary attacks. They may not be practical in a Windows context and mostly not designed for a dynamic and closed-source binary lifecycle. Furthermore, policy-based mechanisms may be less practical as it assumes users can create and maintain complex policies. This is not realistic in commodity operating systems like Windows. Thus, a good tradeoff between security and usability is needed.

Our goal is to increase security in the binary lifecycle. We propose and formalize a family of security models, *BInt*, which provides binary integrity and protection by incorporating an easy-to-use trust labelling mechanism. We also propose FInt which extends to protecting other critical files. We apply BInt by implementing a Windows prototype which protects against common binary attacks while giving good compatibility with existing software and deals with binary lifecycle issues. It is fairly easy to use without the need for complex administration or policy specification.

1.1 Related Work

The Biba model [11], is an early security policy to secure information flows. Data integrity is achieved by preventing information flow from low to higher levels (labels). However, Biba style models are not suitable for the binary lifecycle problem and it is unclear whether binaries are data or subjects.

Domain and Type Enforcement (DTE) [10] is representative of MAC access control approaches where a policy specifies what access is allowed by domains (states of processes) and types (resources). DTE and also other policy based approaches, e.g. Biba, have usability challenges – how to create and maintain policies dealing with the binary lifecycle given that the software and lifecycle details may be unknown and not under one's control, e.g. Windows.

Signed binaries only allow signed binaries to be loaded or executed [9, 13] However, signing is primarily about establishing trust relationships. It only ensures that the signed binaries are from a party having the key. Requiring all software to be signed is best under a closed ecosystem, e.g. iOS, but less practical in an open ecosystem like Windows.

The Windows binary lifecycle also requires updates – creating new problems for trust management. The security of signing is based on trusting the signing keys, e.g. the Stuxnet worm has a driver signed by a Realtek key, thus, is implicitly trusted by Windows. Revocation checking is expensive as it cannot be

done locally and may not be timely. Bit9 [3] a binary whitelisting system was attacked to compromise systems protected by Bit9 signatures [2].

Self-signed executables [21] is proposed for easier management of software updates for signed binaries. While it protects binary integrity for updates, it does not prevent new malware from being introduced. Deletions cause a problem – the file stubs increasing monotonically over time. It also modifies the normal POSIX file semantics which may break compatibility.

Isolation lets untrusted programs read the trusted system while confining the modifications [15, 16]. Some processes can be executed in isolation domains while others are executed normally in the base system. This may not be practical in the binary lifecycle for software on Windows and the implicit "all" sharing.

While these works provide integrity or restrict binary usage, they are less suited for complex, dynamic and closed-source binary lifecycle environments.

2 BInt Integrity Models

When a program executes, typically, it runs a binary (the executable) which may load other binaries implicitly (dynamic loading of binaries) or explicitly load binaries during execution. We want to protect "unwanted" binaries from being executed or loaded, i.e. protect against the large binary attack surface of Windows [12, 14, 19]. In the binary lifecycle of software, binaries are loaded/created/modified/deleted. Securing binaries requires preventing arbitrary modification/deletion of binaries while allowing some software in the lifecycle to do so.

An important consideration is that many security mechanisms rely on explicit policy specification, e.g. Biba [11] or DTE [10] assume someone creates and maintains the policy. In practice, this assumption may not be workable – users cannot be expected to deal with complex policies. The software environment is often dynamic. Users expect to be able to install, update and uninstall (arbitrary) software (within limits). Software updates and auto-updates must be handled. With closed-source software, the workings of the dynamic lifecycle is not known making policies requiring such details problematic. In practice, an implicit requirement is also compatibility with existing software and its lifecycle.

We propose *BInt* (Binary INTegrity) which is a family of security models for binary usage and integrity to protect against attacks on the binary lifecycle which takes into account the above considerations.[3] The following examples illustrate the problems BInt models handle. In the Safari Carpetbomb attack [8]: the Safari browser automatically downloaded files onto the user desktop, while Internet Explorer by default allowed DLLs to be loaded by filename instead of a full path. A malicious website can then perform a "binary planting attack" [7] where Safari downloads a malicious DLL which Internet Explorer loads. However, running or loading binaries from the desktop is normal behavior in Windows, preventing this also breaks normal functionality. In the PDF embedded executable attack [5], a malicious PDF file contains an embedded executable, viewing the PDF

[3] A short paper briefly describes a basic form of BInt [20].

action	$L_m(p)$	requires	result	rule
The BInt Model and Rules				
p create f	d-mode	true	$L_d(f) := \perp$	1
	t-mode	true	$L_d(f) := \perp$	2
	i-mode	true	$L_d(f) := L_d(p)$	3
p read f		true		4
p write or delete f	d-mode	$L_d(f) = \perp$		5
	t-mode	$L_d(f) = \perp$		6
	i-mode	$L_d(f) = \perp \vee$ $L_d(f) = L_d(p)$	$L_d(f) := L_d(p)$	7
p load f	d-mode	$L_d(f) \neq \perp$		8
	t-mode	true		9
	i-mode	true		10
p execute f	d-mode	$L_d(f) \neq \perp$	$L_m(p') :=$ d-mode	11
	t-mode	true	$L_m(p') :=$ t-mode	12
	i-mode	true	$L_m(p') :=$ i-mode; $L_d(p') := L_d(p)$	13
p modetrans		authentication	change $L_d(p)$ and $L_m(p)$	14
Rules for BInt+tr$\vee$ and BInt+tr$\wedge$				
p load f	d-mode	$L_d(f) \neq \perp \vee T(f)$		$8_{tr\vee}$
p execute f	d-mode	$L_d(f) \neq \perp \vee T(f)$	$L_m(p') :=$ d-mode	$11_{tr\vee}$
p load f	d-mode	$L_d(f) \neq \perp \wedge T(f)$		$8_{tr\wedge}$
p execute f	d-mode	$L_d(f) \neq \perp \wedge T(f)$	$L_m(p') :=$ d-mode	$11_{tr\wedge}$

Table 1. The BInt Model (**R1-14**) and BInt+tr (**R8**$_{tr\vee}$**, 11**$_{tr\vee}$**, 8**$_{tr\wedge}$**, 11**$_{tr\wedge}$) where $T(f)$ means f is signed by the trusted signature repositories.

runs Javascript to write out an executable which can be run from the PDF. However, a legitimate installer also behaves in this fashion.

We start with a basic BInt model using the following abstractions. We denote in an operating system, the following system entities: processes by p and files by f. Each process and file has associated security labels which represent information associated with the process/file – the notation $L(o)$ denotes the security label of the system entity o.[4]

Processes interact with files and other processes through the actions: *create*, *read*, *write*, *delete*, *execute*, and *load* a file. The *load* action denotes that a process loads a binary file to be used as a DLL. For Windows, we can use our abstraction to model process creation as follows: a process p *executes* a binary to create a new process p'.

BInt uses two kinds of labels, L_m and L_d. A process has a state which we call an *execution mode*. The execution mode label of process p is denoted by $L_m(p)$ which can take three values: *d-mode* (*default mode*); *i-mode* (*install mode*); or *t-mode* (*temporary trusted mode*). Intuitively, d-mode corresponds to the normal (default) execution mode for running software and processes start in d-mode.

[4] For simplicity, directories and threads are not modelled but are easy to add.

Installing/updating software occurs in i-mode. For special cases, t-mode handles scenarios when we want to run software which needs to dynamically create and load binaries but is not meant to be software installation, e.g. building binaries in an IDE or for dynamic temporary binaries created by a process.

The second kind of label on a process p or file f, denoted by $L_d(p)$ or $L_d(f)$ respectively, can be thought of as a *software domain*. Intuitively, a particular software domain labels all the processes and files related to a particular installed software. For example, the software domain could denote the name of a particular software or the software vendor. There is a distinguished software domain $\perp$ denoting binaries which do not have a valid software domain, we call such binaries *b-invalid*, otherwise a binary is *b-valid*. The only relation among software domains is equality ($=$).

Our first BInt model is formalized in rules 1-14 from Table 1. Each rule specifies the requirement and result of an operation on binaries for a given mode. An actual implementation would distinguish binary files from other files (see Sec. 3) but we omit binary tests to avoid cluttering up the model. Throughout the paper, for brevity, we refer to rule i as $\mathbf{R}i$.

A b-valid binary can only be created in i-mode with the software domain of the process creating the binary (**R3**), otherwise only b-invalid binaries are created (**R1**–**2**). File reads are not affected by BInt (**R4**). This helps compatibility. Rules **R5**–**7** deal with binary integrity. To ensure binary integrity, a binary can only be written to or deleted in i-mode if it is b-valid with the same software domain or if it is b-invalid (**R7**). The integrity of b-invalid binaries is not maintained, so there are no restrictions in d/t-mode as long as it is b-invalid (**R5**–**6**).

Rules **R8**–**13** deal with the use of binaries (load/execute). In d-mode, only b-valid binaries can be loaded/executed (**R8,11**). In t-mode and i-mode, any binary can be loaded (**R9, 10**) and executed. In our abstraction (as in Windows), executing a binary creates a new process. The execution mode of process p is preserved in the new process p' (**R11-13**) and in i-mode, the software domain of p carries to p' (**R13**).

A process changes its execution mode from d-mode to either i-mode or t-mode through a special operation, called *modetrans*. Changing d-mode to i-mode changes both the mode and software domain of the process, while changing to t-mode only changes $L_m(p)$ as the domain is not used (**R14**). `Modetrans` is a privileged operation, for example, it could be implemented with a secure authentication mechanism requiring a password to the operating system. `Sudo` in Unix or UAC in Windows also require secure authentication but they elevate privileges which `modetrans` does not.

Unlike policies where labels are explicitly specified, our labels on processes and files are implicit. In d-mode and t-mode, file labels are implicitly created as $\perp$ and its process label is not relevant. `Modetrans` allows d-mode to go into i-mode. When switching to i-mode, a label is specified which is the software domain used to label the process. File labels in i-mode come from the software domain obtained from modetrans. In terms of user interaction, the user only specifies the software domain once when performing the privileged `modetrans` operation.

`Modetrans` can be thought of as a simple way of associating trust relationships between binaries and its label where the labelling is automatic using just the software domain label from the d-mode to i-mode transition.

Installing and updating software in i-mode assumes that the installer/updater process(es) are part of the process tree hierarchy from the original process in i-mode for that software domain. While this is reasonable for a generic model, it needs to be customized for a particular operating system – in Windows, we handle the Windows MSI installer and provide an execution mode policy for auto-updaters (see Sec. 3).

2.1 Using BInt

We use a life-cycle of the Firefox web browser to illustrate how BInt works. The user first downloads the Firefox installer ($f_{installer}$) using some other web browser or downloader ($p_{downloader}$), which runs in d-mode ($L_m(p_{downloader}) =$ d-mode). By **R1**, $L_d(f_{installer}) = \perp$. The user then uses the privileged *modetrans* operation to run the installer in i-mode specifying its software domain as `firefox`. The installer process ($p_{installer}$) and its child processes run in i-mode with the `firefox` domain, $L_m(p_{installer}) =$ i-mode $\wedge L_d(p_{installer}) =$ `firefox`. The installer installs a number of binaries, which are in the `firefox` domain according to **R3**. After installation finishes, the user executes Firefox from the Windows start menu or desktop shortcut. At this point, the Firefox process runs in d-mode due to **R11**.[5] Suppose that Firefox is exploited by a malicious website, e.g. a drive-by-download downloads and runs a malicious executable (f_{mal}). However, the binary f_{mal} has $L_d(f_{mal}) = \perp$ by **R1**, thus, f_{mal} cannot execute by **R11** and the attack fails.

In order for Firefox's auto-update to work, the updater is specified in the execution mode policy (see Sec. 3) so that it automatically runs in i-mode with `firefox` domain. To uninstall, the user uses `modetrans` to execute the uninstaller, which then deletes the Firefox binaries without affecting other binaries by **R7**.

A different scenario occurs during software development – the programmer is often creating binaries which may be transient. The IDE can be run in t-mode allowing the software developed to be temporarily executed (**R9**, **R12**).

2.2 BInt+tr: Adding Further Trust

BInt focuses on maintaining integrity of binary files. Which binaries to trust is an orthogonal issue. In accordance with defence in depth, we extend BInt with an additional source of trust. We assume an external trusted signature repository publishing signatures of vetted binaries, e.g. such as Bit9 [3], but alternative

[5] We assume the user is familiar with the usage and principles of BInt. The user should not launch Firefox from the installer since normal software execution should be in d-mode. However, similar to Windows UAC prompts, warnings can be issued when executing a new binary in i-mode.

mechanisms are also possible. The binary signature is used (additionally) to certify that a binary and associated software domain is trusted.

We describe two alternative models, BInt+tr$\vee$ and BInt+tr$\wedge$ presented in Table 1 from $\mathbf{R8}_{tr\vee}$ to $\mathbf{R11}_{tr\wedge}$. BInt+tr$\vee$ is a more permissive model which allows binary f to be loaded/executed in d-mode if f is b-valid *or* if it is certified by the trusted signature repository ($\mathbf{R8}_{tr\vee}$,$\mathbf{11}_{tr\vee}$). For example, third parties can certify a list of trusted software, a user can use the list to avoid switching to i-mode to install software in the list. This allows for broader software compatibility without compromising the integrity of other binaries. It also reduces the use of i-mode but requires a trusted service.

A restrictive policy is BInt+tr$\wedge$ which requires both signature verification and b-validity ($\mathbf{R8}_{tr\wedge}$,$\mathbf{11}_{tr\wedge}$). For example, in an organization, this can enforce that only specified software can be used and exceptions only occur through t-mode. Variations of the signing requirements for t-mode are also possible. The whitelist approach of BInt+tr$\wedge$ may be too restrictive for general use since only binaries on the whitelist can be executed. A practical incarnation may only require the verification for certain pathnames of software domains. Incorporating an external trust mechanism allows to add MAC policies and also an ecosystem of security providers which provide whitelists of vetted binaries, e.g. similar to Bit9 [3].

An even more restrictive form of BInt+tr$\wedge$ is to require that binaries created in i-mode must pass the signature verification, otherwise, the creation and subsequent writing of the binary has no effect.[6] We call this variant, BInt+tr$\wedge$W.

2.3 Analysis of BInt Models

The binary protection from BInt arises in three ways. First is whether execution or loading of binaries is prevented. Note that this does not prevent all malware code execution, e.g. code injection, we focus on attacks employing binary mechanisms. Second is it provides integrity guarantees for binaries, preventing malware from modifying binaries. Thirdly, in order for malware to persist on the system, it will normally need to be in files (binaries), otherwise the vulnerability must be one which can reoccur on the same system which we do not deal with.[7] We remark that without tailoring BInt for a particular operating system, execution/loading/reading/writing of binaries are the only relevant operations in our model when dealing with binary files so the discussion focuses on these operations and also `modetrans`.

Security of **d-mode**: Most processes run in d-mode, thus its security is critical. The threat model is whether a process in d-mode can execute/load an undesired binary (b-invalid binary) or modify existing (b-valid) binaries.

[6] This changes the semantics of file write so that changes behave like a shadow file until it can be verified when the file is closed. Self-signing [21] also needs to work in a similar way.

[7] E.g. a vulnerability in the network code in the operating system might allow an attacker to gain arbitrary code execution within the kernel with an external network request, however, this is not a binary vulnerability or exploit.

The guarantees in d-mode are: b-invalid binaries cannot be loaded (**R8**); b-valid binaries cannot be modified (**R5**); and binaries created are b-invalid (**R1**). Thus, a d-mode process is unable to introduce new binaries to d-mode processes including itself which prevents common attacks which use execution or binary loading. This prevents both the example attacks (Safari Carpetbomb and PDF embedded executables). The integrity guarantee is that existing binary files which are b-valid cannot be modified by the attacker. It is also not possible to delete b-valid binaries. As d-mode is orthogonal from other privileges, i.e. system administrator privileges, these guarantees apply even for privileged processes in d-mode. An important consequence is that even if a software running in d-mode is successfully attacked, the attack cannot be made *persistent* through binaries as it cannot write b-valid binaries nor can it modify any binaries. Since the operations considered on binaries are execution, loading and file operations, this completes the analysis of d-mode.

Security of **i-mode**: Changing from d-mode to i-mode using `modetrans` requires authenticated privileges for the operation (**R14**), thus, no processes in d-mode can enter i-mode by themselves. So the threat model is that an attacker needs to get the user to enter i-mode, e.g. a social engineering attack. However if BInt+tr$\vee$ is used, then `modetrans` can be a rare and unusual operation making is more difficult to social engineer unlike UAC in Windows where the user is "trained to click allow".

There are two cases to consider whether the user installs the malware in a new software domain or existing domain. Firstly, if it is a new domain, as the malware installer cannot modify existing b-valid binaries, their integrity is assured. However, the malware can install new binaries which might be loaded into existing software, e.g. the Safari Carpetbomb DLL attack, if there is an exploitable vulnerability. The BInt+tr$\wedge$ model (and BInt+tr$\wedge$W) can prevent this since the malware should not be in the whitelist. Our prototype additionally keeps a *binary database* of binaries and their software domains and also logs of binary usage and loading relationships, allowing attacks to be detected and be removed more easily. Secondly, if it is an existing domain, the malware can modify binaries of the above BInt models except in the BInt+tr$\wedge$W model which only allows modification with another trusted binary of the same domain. Thus, BInt+tr$\wedge$W being the most restrictive model prevents binary integrity attack in both cases.

The damage that can be caused by the malware in the other BInt models depends on the domain. For critical domains such as `microsoft` (for all the system binaries), the malware can affect all software as programs use Windows system DLLs in the `microsoft` domain. To reduce the impact of such attacks, one approach is to require extra privileges such as a separate password for critical domains. Furthermore, unlike the Windows UAC privilege escalation, the binary database in the prototype can be used to explain whether a binary is relevant to the software domain.

The extensions discussed in Sec. 5 also reduce the threats from i-mode.

Security of **t-mode**: Like i-mode, t-mode also requires authentication for the privilege escalation. However, t-mode behaves like d-mode in terms of binary integrity, a t-mode process cannot modify b-valid binaries (**R6**). Thus, a malicious t-mode process cannot introduce new binaries to d-mode processes. However, t-mode processes can load b-invalid binaries (**R9**) which allows for binary attacks to these processes. T-mode is meant to be a special exception, it is like i-mode in that most software and processes do not run in this mode. Since in t-mode, any binary can be loaded, the threat model is whether the attacker can make the malware persist. However, in order to persist, it would need to be able to lure the user to authenticate and run it in t-mode every time, as it cannot execute/load in d-mode and t-mode does not affect binaries. We argue that unlike UAC, user authentication for i/t-mode is more controlled and without the problem that users tend to choose "always allow" [17]. The problem with UAC is that users do not know how to choose between allow or deny, they learn that deny just means the software fails, so they learn to click "allow".

3 A BInt Windows Prototype

We implemented a prototype in Windows XP of BInt models. We describe the implementation of basic BInt and mention differences for other models. We also discuss some implementation features for our models to deal with special features in Windows as BInt is generic and the model is not targeted for Windows. We use a kernel driver in Windows XP to intercept native calls (Windows system calls) for binary loading, file reading, file modification, process creation and some other operations.[8] It also maintains the labels of processes and binaries. As our implementation works inside the Windows kernel, it allows us to apply all the rules of BInt to all processes and binaries in Windows.

Our prototype is meant to be a proof of concept to show that BInt can be implemented efficiently, provide security against binary attacks and be compatible with existing software. Nevertheless, the prototype shows the viability of BInt and that it would be relatively easy for Microsoft to implement. It should be clear also that building a version of BInt in another operating system, e.g. Unix, is relatively straightforward.

For **R8**–**13** in Table 1, we intercept the `NtCreateSection` native call, which is necessary for binary loading. If the execution mode is d-mode and the binary is b-invalid, `NtCreateSection` fails resulting in the load/execute failing. For **R1**–**7**, we intercept the `ZwCreateFile` call, which opens or creates a file and returns a handle. For **R11**–**13**, we use the kernel API `PsSetCreateProcessNotifyRoutine` to inherit execution mode and software domain in the child process. For **R14**, we use IOCTL (I/O control) to implement the system call-like `modetrans` operation.

Most of the corresponding rules in BInt+tr are implemented in the same way as BInt. The $T(f)$ signature verification in BInt+tr is cached so that multiple loadings only need a single verification unless the binary is modified. This caching

[8] We use Windows XP, later versions require signed drivers.

optimization is similar to that in [13] which has been shown to be efficient with negligible overhead for real applications. File writing, renaming and deletion are monitored through the `ZwCreateFile` and `ZwDeleteFile` kernel APIs.

We assume all file modifications are under the control of the operating system kernel. This assumption can be invalid in some cases. When the system mounts a network shared file system, (e.g. through SMB) an attacker can change the binaries outside the system. Similarly, files can be changed when the system is offline. We call such files, *unmonitorable files*. To prevent these attacks, we use file signatures to detect modification. A *binary database* stores information about files, signatures, modification history and other metadata. We also generate logs of how binaries were used which is useful for explanations and creating special exceptions, e.g. execution mode policy. Log maintenance is done outside the kernel. For unmonitorable b-valid files, their signatures are updated immediately after the file is modified. For binaries that just come online, we verify the signatures once for each binary and cache the result [13]. We optimize the signature verification with a lazy way of updating signatures to reduce the overhead of signature verification. We store normalized internal kernel paths to disambiguate Windows 8.3 filenames, long file names and symbolic links. For the NTFS filesystem, we use the object ID to disambiguate hard links.

In Windows, there is no distinguishing feature of a binary (the filetype is only a convention, i.e. an executable need not have file type `.exe` or `.com`), other than its format. We test whether or not the file is a binary by reading the file header. This makes i-mode more costly than other modes since only i-mode creates b-valid binaries. We modify the semantics of Windows slightly so that files opened for writing in i-mode are in exclusive mode to simplify signature creation. We do not expect this to be a major restriction as the installer is likely to be creating files sequentially. When p closes the file handle of f, the file contents is now complete and f's signature can be re-computed (lazily). In principle, signatures only need to be maintained for non-monitorable files. However, to reduce the impact of offline attacks, we choose to maintain signatures of all b-valid binaries and critical files.

Since a software installer may launch several helper programs to accomplish the installation, we need to ensure all helper processes are labelled with the same execution mode and software domain. This is accomplished by mode and domain inheritance (**R11**–**13**). BInt assumes that all helper processes are the child (or descendant) processes of the first installer process. While the assumption holds for most installers, there is an important exception. MSI (Windows Installer) is a generic installation engine for installing and updating software on Windows. It is used for both Microsoft and non-Microsoft software. MSI makes use of a service (daemon) process to perform installation. The service process is always running and is not part of the process hierarchy of the original installer. Dealing with MSI requires some minor extensions to how i-mode works. We monitor the communication channel, a named pipe `\Pipe\Net\NtControlPipeX`, between the installation process and the MSI service. When an i-mode process triggers the service to start installation, the service is switched to i-mode with the same

domain as the triggering process. When the installation terminates, the service is switched back to d-mode. As the MSI service is used atomically, there is no interference between concurrent requests.

We now illustrate how BInt is used in our Windows prototype. When the system is booted, the initial process(es) run in d-mode, thus all subsequent processes run in d-mode unless `modetrans` is used. We have implemented a command-line `modetrans` utility which authenticates the user using a password and executes a user-specified program in a user-specified mode and domain.

Since auto-update program should always run in i-mode, we introduce an *execution mode policy* which simplifies system usage by predefining special cases where the operation of `modetrans` can be performed automatically. For example, Windows auto-update (`wuauclt.exe`), is specified to run in i-mode and the `microsoft` domain. Finally, the mechanisms also protect the BInt policy files, modifications to the policies require user authentication.

The execution mode policy is to make usage of BInt more transparent so that users do not need to explicitly go into i-mode. This policy is small and mostly for well-known cases with a few exceptions. Thus, it is much easier to deal with and maintain than more complex policy-based models.

3.1 Evaluating BInt on Windows

We evaluated our prototype with the basic BInt model as the other models would need additional external trusted third party providers. In terms of performance, the main mode is d-mode as other modes should only be used more rarely. As we employ caching to monitor binaries, once a binary signature has been checked, there is little overhead (as the implementation of signed binaries in [13]). Since many binaries are shared, we find that once the system has started and some binary has been loaded before, the overheads for real applications are negligible and we did not notice any significant difference between running our prototype and normal Windows.

We evaluated d-mode on the following common binary attack vectors: directly running a b-invalid executable from the GUI Windows Explorer shell (a social engineering attack) and the `cmd` shell; PDF attack on Acrobat Reader using a PDF embedded binary [5]; loading a b-invalid driver; starting a b-invalid service; loading b-invalid shell extensions and Browser Helper Objects (exploits a vulnerability where binaries could be loaded as a Windows help file [4]); and loading b-invalid DLLs by `PATH` manipulation (such as DLL planting attacks [7, 8]). While our security analysis already shows that d-mode prevents these attacks in the abstract model, the evaluation confirms this for the prototype.

We tested compatibility with the binary lifecycle of common Windows software by installing, running and uninstalling the following applications: Internet Explorer (IE, highly integrated into Windows), Winamp (music player with 88 binaries), Yahoo Messenger (instant messaging client with 55 binaries), Firefox (32 binaries), Google Chrome (137 binaries), Adobe Acrobat Reader (31 binaries) and Java Development Kit (229 binaries).

IE tests Microsoft software installation. The software domain is `microsoft` due to the highly integrated nature of Microsoft software in Windows. In fact, IE modifies several Windows system DLLs. No problems were observed during installing and running IE. Windows update handles the auto update of Windows related software including IE, this occurred transparently without problems.

The Winamp installer uses its own Nullsoft installer. No problem was observed during running and uninstalling Winamp. Yahoo Messenger uses a network-based install, the installer is an initial installer which downloads a much larger installer. The installer tries to upgrade the Flash ActiveX plugin `flash.ocx` if it is out of date. This action is blocked as the software domains do not match. However, this is not a problem as the Flash plugin can be updated separately. We noticed that a `YahooAUService.exe` service is created for auto-update. In order for the auto-update to work transparently, we should add `YahooAUService.exe` to the execution mode policy to run in i-mode with the `Yahoo` domain.

No problem was found during Firefox installation. Auto-updates are handled by `updater.exe` in the Firefox software domain. For transparent update, it is added to the execution mode policy. No problem was observed for Chrome. Reader and Java Development Kit use the MSI engine which is handled transparently without any user interaction.

We tested typical software which cover a range of mechanisms for installation, uninstall, and update. We found that usage scenarios for the software lifecycle aspects are usable with little effort needed. In some cases, the security policy achieves complete transparently. For full transparency, the execution mode policy is used with a minimal specification. This can be done manually immediately after installation if the user knows which program does the update. It can also be done at the first time the updater performs the updates. In this case, the user will be notified about the attempt to modify binaries. Information from the binary database and logs can then be used to set the execution mode policy. Alternatively, auto-updaters can be run manually in i-mode as a more secure alternative which does not rely on any execution mode policy specification. Naturally that requires a bit more effort on the part of the user. If a more secure policy is needed we should expect that it needs some information but it should be sufficiently easy to specify and maintain without extensive analysis and expert knowledge, which is how we designed BInt.

4 The FInt Model

BInt only covers binaries but the integrity of non-binaries may also be important for the security of the system. For example, the attacker can modify the Java class files used by the Java compiler to insert malicious bytecode even though neither the Java virtual machine nor compiler is compromised [18]. An attack can modify a good script (`.bat`) into a malicious one. Without modifying the web server binaries, the attacker can change the web server's configuration file or PHP script to steal data or modify the web site.

action	$L_{m'}(p)$	requires	result	rule
p create f	d-mode$'$	true	$L_d(f) := \bot$	1_f
	t-mode$'$	true	$L_d(f) := \bot$	2_f
	i-mode$'$	true	$L_d(f) := L_d(p)$	3_f
p read f	d-mode$'$	α (see caption)		4_f
	t-mode$'$	true		5_f
	i-mode$'$	true		6_f
p write/delete f	d-mode$'$	$L_d(f) = \bot$		7_f
	t-mode$'$	$L_d(f) = \bot$		8_f
	i-mode$'$	$L_d(f) = \bot \vee L_d(f) = L_d(p)$	$L_d(f) := L_d(p)$	9_f
p modetrans		authentication	change $L_d(p)$ and $L_{m'}(p)$	10_f

Table 2. Rules for FInt. Assumes the files are non-binary. α: Apply the FInt policy for pathnames. If the result is "verify", the condition is: ($L_d(f) \neq \bot$ (when owner flag is not set)) $\vee$ ($L_d(f) = L_d(p)$ (when owner flag is set))

We generalize BInt to protect integrity of any file. The use of files which are not binaries is quite different from binaries. Firstly, the operating system does not distinguish between an interpreter executing a script and reading a data file. Usually we only want to protect the integrity of the former. Secondly, while there are usually many more non-binaries than binaries, only a small fraction of the non-binaries is critical to security of the system. Thirdly, the semantics of non-binaries is program specific, unlike binaries which the operating system understands, e.g. a malicious Perl script is significant when opened by the Perl interpreter, but not when opened by a text editor.

Due to these differences, we adopt a different approach to protecting the integrity of non-binaries. Only files defined by a *FInt policy* are protected. Essentially the FInt policy specifies what pathnames are critical to certain software. This policy can be specified on a per-program or per-domain basis. We remark that other variants of FInt are possible, we present FInt as a conservative extension of BInt.

The FInt policy consists of a list of subjects. Each subject is associated with a list of objects (pathnames) and associated action. The subjects and objects correspond to processes and files in the operating system. The subject is defined by a pathname of a binary or a software domain; and the object is a rule for the subject defined by a regular expression for a pathname along with the following actions: *allow*, *deny*, or *verify*. *Allow* means the files are allowed to be read/loaded. Files matching the *allow* rule are considered to be not critical to the program. *Deny* means the files are denied from being read/loaded. *Verify* means that the reading/loading is allowed depending on the execution modes and software domains of the process and file. The FInt policy extends BInt (specifically, rules **R4**, **R11**-**R13**) by applying to all files including binaries.

More than one rule can be specified for a subject. The action specified by the first matching rule is taken. The default action (none of the regular expressions match the path) is *verify* for binary and *allow* for non-binary. This is to make FInt consistent with BInt when no FInt policy is specified.

In FInt, non-binaries are labeled with software domains similar to BInt. The default label is $\perp$ for *all* files unless otherwise created with a different domain which extends the notion of b-invalid ($L(f) = \perp$) and b-valid ($L(f) \neq \perp$) to all files. We introduce file execution modes in FInt which add to those in BInt. New file execution mode of a process p are denoted by $L_{m'}(p)$, namely: d-mode$'$, t-mode$'$, and i-mode$'$. Table 2 formalizes FInt for non-binary files. We add a *file execution mode policy* which specifies which programs should be automatically executed in which file execution modes. For example, the Java compiler can run automatically in i-mode$'$ so that the compiled class file will be b-valid, and is unmodified by anything else when used by the Java VM.

In order to prevent a program from (accidentally) reading files created by other programs (e.g. malware), we introduce an optional flag *owner* for each policy rule – the flag means that the file read/loaded must not only be b-valid, but also have the same software domain as the process (see Ex3).

One motivation for FInt policies is that they can be used to construct specialized behavior for FInt. It can also be used to create special security policies or restrictions. We give some examples of how to use FInt policies.

Ex1: The following simple policy protects all batch files for the CMD shell:

```
[c:\windows\system32\cmd.exe] verify .*\.bat
```

Ex2: The following policy verifies Java bytecode coming from `.class` and `.jar` files except for a project directory. The purpose is to allow modification of the Java code under development by non-JDK program such as IDEs. This policy is shared by programs such as `java.exe` and `javaw.exe` (GUI version of `java`) in the JDK software domain.

```
[jdk]
  allow  E:\\projects\\foo\\.*\.class
  allow  E:\\projects\\foo\\.*\.jar
  verify .*\.class
  verify .*\.jar
```

Ex3: The following policy prevents Firefox's built-in JavaScript modules and extensions from being hijacked by third party program. It uses the "owner" flag.

```
[firefox]
  verify owner C:\\Program Files\\Firefox\\.*\.jar
  verify owner C:\\Program Files\\Firefox\\.*\.js
  verify owner C:\\Program Files\\Firefox\\.*\.xul
```

Ex4: In order to prevent the web server from being exploited to launch a `cmd` shell, one may run the web server in a more restricted environment with `cmd.exe` blacklisted by the following policy. Even without this policy, the web server is already protected as binaries which are not b-valid cannot be executed. Thus, if an attacker breaks in, they cannot run their own binaries but are restricted to the existing b-valid binaries. This policy further reduces the allowed binaries by denying the `cmd` shell.

```
[apache] deny .*\\cmd.exe
```

5 Discussion & Conclusion

We discuss further extensions and possibilities for BInt for which there is lack of space to go into the details.

File Deletion: For simplicity, the uninstaller runs in the same execution mode, i-mode, as installer and updater. This allows the uninstaller to add new binaries. We can prevent this by introducing a *u-mode*, which is more powerful than d-mode and less powerful then i-mode. In u-mode, the process can delete binaries with the same software domain or $\perp$, so that it can delete its binaries. Binaries created by a u-mode process are $\perp$, so that it cannot introduce new binaries. We remark that as an alternative to deletion, the label can simply be downgraded to $\perp$ if the binaries are to be retained but not executable or loadable.

Software Dependencies: In BInt, software domains are treated equally, i.e. a process running in one software domain can load a binary of another software domain. If a user accidentally installs a malicious binary, it can be loaded into all processes. To prevent this, we can incorporate software dependencies into BInt, so that a process can load a binary if the software group of the process depends on the group of the binary. This adds a partial order relation while BInt only needed equality. For example, the dependencies can specify a plugin of a web browser can only be loaded by the browser but not other software. The dependency information can either be specified during software installation or come from a trusted third party such as the software developers.

Sandboxed Domains: BInt requires i-mode to first install software before use, which may be considered troublesome to some users, i.e. users expect to be able to run a software immediately after downloading. We can use the idea of a *sandboxed domain* to allow immediate execution while still prevent the new binary from being loaded by other software. Binaries created by d-mode or t-mode process are assigned a new sandboxed software domain (instead of $\perp$ in BInt) – the new domain is denoted by *newsb*. Any binary can be executed in d-mode but if the binary is from a sandboxed domain, the process label also becomes sandboxed (a modification of **R11**). Thus, a downloaded binary can be executed immediately. However, to prevent malware from automatically executing downloaded binaries, when a sandboxed domain is executed for the first time, a UI prompt (with the creator's software domain, creation time, binary path, etc.) will ask for permission.

In summary, we have proposed a flexible family of binary integrity models which are designed to handle dynamic creation, modification and deletion of binaries in their lifecycle. Our models combine integrity of the binaries together with trust to protect against typical attack vectors which exploit the use of binaries which is a major headache in Windows. Our models are suitable as a security enhancement for Windows since the large attack surface of Windows leads to many binary attacks which BInt models prevent. Our prototype demonstrates that these models are practical and easy to use. As binary attacks are commonplace in Windows, we believe what is needed are simple policy mechanisms which give a good tradeoff between usability and security. BInt does not deal

with code injection attacks but it can be combined with other runtime security mechanisms which do that, e.g. ASLR, NX, etc.

While we have focused on Windows, the BInt models are general and can be applied to other operating systems. Although Windows is where BInt would have the biggest benefit, there are also documented attacks on Unix such as autorun-style USB attacks in Linux [6] and the Flashback and Mac Defender malware on Mac OSX. We also propose FInt as a conservative extension of BInt to protect the integrity of non-binary files. We believe that the recent ShellShock bug in `bash` (a script injection vulnerability regarded as critical in most Unix/Linux systems) can be mitigated with extensions to our models.

References

1. HEUR:Backdoor.Java.Agent.a
   `https://www.securelist.com/en/blog/8174/A_cross_platform_java_bot`
2. `http://www.computerworld.com/s/article/9237295/Researchers_link_latest_Java_zero_day_exploit_to_Bit9_hack`
3. `http://www.bit9.com`
4. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0483`
5. `http://blog.didierstevens.com/2010/03/29/escape-from-pdf`
6. USB Autorun Attacks in Linux
   `http://blogs.iss.net/archive/Shmoocon2011.html`
7. `http://www.microsoft.com/technet/security/advisory/2269637.mspx`
8. `http://www.oreillynet.com/onlamp/blog/2008/05/safari_carpet_bomb.html`
9. A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi and V. Roy, DigSig: Run-time Authentication of Binaries at Kernel Level, USENIX LISA, 2004.
10. L. Badger, D.F. Sterne, D.L. Sherman, K.M. Walker, and S.A. Haghighat, Practical domain and type enforcement for UNIX, IEEE S&P, 1995.
11. K. Biba, Integrity Considerations for Secure Computer Systems, ESD-TR-76-372, MITRE, 1977.
12. T. Dai, M. Zhang, R.H.C. Yap, and Z. Liang, Understanding Complex Binary Loading Behaviors, ICECCS, 2014.
13. F. Halim, R. Ramnath, Sufatrio, Y. Wu and R.H.C. Yap, A Lightweight Binary Authentication System for Windows, Joint iTrust and PST Conf., 2008.
14. M. Howard, J. Pincus, and J.M. Wing, Measuring Relative Attack Surfaces, Workshop on Advanced Developments in Software and Systems Security, 2003.
15. K. Kato and Y. Oyama, SoftwarePot: an Encapsulated Transferable File System for Secure Software Circulation, Intl. Symp. on Software Security, 2003.
16. Z. Liang, W. Sun, V. Venkatakrishnan and R. Sekar, Alcatraz: An Isolated Environment for Experimenting with Untrusted Software, ACM TISS, 12(3), 2009.
17. S. Motiee, K. Hawkey, K. Beznosov, Do Windows Users Follow the Principle of Least Privilege? Investigating User Account Control Practices, SOUPS, 2010.
18. K. Thompson, Reflections on Trusting Trust, CACM, 27, 1984.
19. Y. Wu, R.H.C Yap and R. Ramnath, Comprehending Module Dependencies and Sharing, ICSE, 2010.
20. Y. Wu and R.H.C. Yap, Towards a Binary Integrity System for Windows, ASIACCS, 2011.
21. G. Wurster and P.C.V. Oorschot, Self-Signed Executables: Restricting Replacement of Program Binaries by Malware, USENIX HotSec, 2007.