Automated Installation Verification

Erik Peter Wibeck
2015

Bachelor of Science
Computer Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Abstract

When installing a software product there is a risk that the installed product does not start after the installation or the software does not work as expected due to missing files or wrong/missing configuration. Also, when uninstalling the product, there is a risk that files and registry keys are left on the system which can lead to system instability. Software manufacturing companies are negatively affected by these issues: if the software does not install properly the end user perception of the product quality will be low regardless of the actual quality of the final product. This can lead to customer dissatisfaction, fewer customers, bad reviews and bottom line: loss of revenue.

This thesis has implemented a stable and predictable solution to verify, in an automated approach, that a software installation has been executed as expected. This means that there are no side effects and no unexpected changes to the operating system. The result shows that the solution vastly improves the testing of a software installer compared to existing methods. The solution has been developed targeting the .NET platform on Windows 7 or newer. By automating the test execution it is now possible to tell the state of the software installer for every build. Monitoring the system in real time does detect all changes and who did the change, hence, all changes by the software installer are accounted for.
Contents

Abstract ........................................................................................................................................................................1

List of figures ..................................................................................................................................................................5

1 Introduction ......................................................................................................................................................................6
  1.1 Background ..............................................................................................................................................................6
  1.2 Motivation .................................................................................................................................................................7
  1.3 Problem definition ......................................................................................................................................................7
  1.4 Delimitations ..............................................................................................................................................................8
  1.5 Thesis structure ..........................................................................................................................................................8

2 Related work ...............................................................................................................................................................9

3 Theory ...........................................................................................................................................................................10
  3.1 Evaluating different techniques to monitor a file system and registry for changes .........................................10
    3.1.1 FileSystemWatcher, ReadDirectoryChangesW and FindFirstChangeNotification API ..........................10
    3.1.2 SHChangeNotifyRegister API .........................................................................................................................11
    3.1.3 NTFS change journal .........................................................................................................................................11
    3.1.4 Snapshot ............................................................................................................................................................12
    3.1.5 Kernel file system filter driver (minifilter) .........................................................................................................12
    3.1.6 API hooking ........................................................................................................................................................14
    3.1.7 Event tracing for windows ...............................................................................................................................16
    3.1.8 Kernel registry filter driver ................................................................................................................................18
    3.1.9 System Registry provider in WMI .........................................................................................................................19
    3.1.10 Process Monitor ..................................................................................................................................................20
    3.1.11 Selecting the way to monitor the system ..........................................................................................................21

3.2 Evaluation of methods to filter event data ................................................................................................................22
    3.2.1 Filter on processes related to the installation ....................................................................................................22
    3.2.2 Filter event based on event type .........................................................................................................................23
    3.2.3 Filter events based on windows installer internal housekeeping ........................................................................23
    3.2.4 Filter events based on timespan ..........................................................................................................................24
    3.2.5 Selecting the way to filter events .......................................................................................................................24

4 Implementation ..........................................................................................................................................................25
  4.1 System overview .......................................................................................................................................................25
    4.1.1 System workflow ................................................................................................................................................25
List of figures

Figure 1: Filter manager I/O stack [23] ........................................................................................................... 13
Figure 2: Normal execution compared to hooked execution ............................................................................ 15
Figure 3: The event tracing architecture [30] ................................................................................................ 16
Figure 4: Process start event ......................................................................................................................... 22
Figure 5: Flow diagram for filtering events .................................................................................................... 24
Figure 6: Verification classes’ pair ................................................................................................................ 26
Figure 7: Filter based on event type ............................................................................................................... 27
Figure 8: Filter based on related process ....................................................................................................... 27
Figure 9: Find files changes workflow ......................................................................................................... 28
Figure 10: Find registry changes workflow ................................................................................................ 29
Figure 11: Install chainer test data file ........................................................................................................ 31
Figure 12: MSI test data file .......................................................................................................................... 32
Figure 13: Properties test data file ................................................................................................................. 34
Figure 14: Screenshot of regedit showing registry key structure .................................................................... 36
Figure 15: Screenshot of regedit showing an integer value ........................................................................ 37
Figure 16: An example log listener ................................................................................................................ 39
Figure 17: Check class example .................................................................................................................... 44
Figure 18: Test data loader example ............................................................................................................. 45
Figure 19: Test data example ........................................................................................................................ 46
Figure 20: Install test execution output example .......................................................................................... 47
Figure 21: Automated test cases example ..................................................................................................... 48
Figure 22: Installation location ..................................................................................................................... 55
Figure 23: Test data definition file ............................................................................................................... 55
Figure 24: Setup configuration ...................................................................................................................... 56
Figure 25: Test case ....................................................................................................................................... 57
Figure 26: Test result output .......................................................................................................................... 57
1 Introduction
This chapter gives a background about software installers and software installation testing. It is also presenting the problem that this thesis aims to solve.

1.1 Background
Nowadays, the number of applications available for any end user to be installed in a computer is bigger than ever, at the same time, computer systems are getting more sophisticated and complex. This means that more users are exposed to low quality installers, risking the integrity of the systems.

The configuration matrix for a software installer is affected by many parameters such as operating system version, 32/64 bit and other already installed components. These all may affect, for example, what files are getting installed or what services are started. Most software development companies provide some type of installation packages to assist in the installation of their applications, but unfortunately, the outcome is not always successful. Especially due to the large number of variables/parameters associated with any specific computer, the installation may appear to complete successfully, while there are mistakes and problems with it. Also, the number of changes to a software installer during “system development life cycle” [1] may be many and can change on a daily basis. “Resources that often change include the program’s executable code, help files, readme and license files, system code, system registry entries, and components installed into other software products to allow for feature integration.” [2]

Installation software testing has been around for a long time as something some software development companies do in the end of the “system development life cycle” [1], before the software is shipped to businesses or individual customers. “Installation testing is a kind of quality assurance work in the software industry that focuses on what customers will need to do to install and set up the new software successfully. The testing process may involve full, partial or upgrades install/uninstall processes.” [3] The installation testing is often done manually, when done at all. For example, the “Chromium Projects” [4] project defines seven test cases to be executed against the UI manually where the expected result to verify is focusing on UI behavior, and not the actual installation result. The “Mozilla installer test plan” [5] from Mozilla and “OpenStack installer test plan” [6] from CISCO are other examples of installer test only focusing on UI behavior and manual verification.

The combination of complexity, change frequency and number of variables affecting the installation, when most companies are doing manual software installation testing, is a mix that makes it hard to ensure the quality of the installer. Some companies have started to automate their software installation testing effort using tools to compare state of machines before and after an installation, which is an unstable and unpredictable method (as explained further on).
1.2 Motivation

Many advance users of computers have at least once tried to install an application just to find out that the installation fails, the application does not start, or the application does not work due to missing files or configuration. Also, when uninstalling an application it is not uncommon to find traces of the application still on the computer, this is not only a waste of storage on the computer but it could also lead to stability problems and inability to install other applications on the system. This is bad for the branding of the company manufacturing the software, independently how good the product is. The end user will never see the product if the installation fails and will be less likely to try another product from the same company again. This should have been solved already as part of the development process.

Searching for the word Uninstaller on any application download site will give hundreds of hits. For example, http://en.softonic.com gives a list of 523 programs (search performed 15-4-2015) trying to solve the problem that installers are not properly tested or tested at all. The common approach for all of these programs is that they take a reactive approach and that they are trying to solve the problem after the fact of a failure.

During my years of working with software installers and specific software installation testing, I have yet not found any product that can automate the installation testing process and produce a predictable test-run result covering both expected and unexpected outcomes. If there was an ‘easy to use’ tool on the market, the above mentioned problems with uninstallation would be easy to solve for a software manufacturer. It would also be possible to run test for every build or change to track the health of the software installation as part of the software development life cycle.

1.3 Problem definition

There does not exist any simple, stable and predictable solution that, in an automated approach, verifies that a software installation has been executed as expected with no side effects.

To build the solution it is necessary to solve the following:

- Find the best method to monitor a system in real time where a software installer is tested on, and then collect the changes.
- Find the best way to reduce noise in collected data to ensure stability and predictability.
- Find a simple way to integrate the solution into any existing or new software installer test automation setup.
- Find a simple way to define test data.

David Maclver describes the problem of not testing the installer in one of his blog posts “I’d manually tested the installer for the 0.2.0 release but when pushing the 0.2.1 release I made a bit of a screw up and included some changes I hadn’t meant to and these broke the installer because they added some new packages without adding them to the setup.py. I forgot to manually test the installer because I thought the change was trivial and voila, one broken package on pypi.” [7] This could have been avoided if he had automated the testing instead.

In the best case a company has created a matrix of different environments where they will install their product by some script to speed up the process. They may even have some scripts that can start the application and validate that it did start. The rest of the validation is normally done by hand and maybe with help of some snapshots to save different stages of the environment to compare later.
1.4 Delimitations
This work is focusing on solving the problem on a Windows 7 operating system or newer, using the Microsoft .NET platform. Windows 2003 and all onward versions have .NET enabled by default, so no additional tools need to be installed [8].

1.5 Thesis structure
Section 2 covers the related work in the area, in section 3 different techniques to monitor system changes and to filter them are presented and evaluated. Section 4 covers the implementation of the solution and integration with the system development life cycle. The thesis ends with evaluation, discussion and conclusion sections.
2 Related work

Greg Christopher has a patent named “Software installation verification” where he describes a method to compare snapshots to detect changes as a result of a software installation. The problem he is trying to solve is detecting unexpected changes between a baseline and a new software installation. “In general, in one aspect, the invention features operations including generating a comparison of a current software installation with a previous software installation, and identifying, based on the comparison, resources that have not changed in their installation result from the previous software installation to the current software installation, despite an expectation that the unchanged resources should change from the previous software installation to the current software installation.” [2].

Rashidah Haron Galoh patent “Automated installation testing and management system” [9] is focusing on two steps: a pre verification step, where the system is validated to be ready for installation; and a post step focusing on performing test when the software has been installed.

Greg Christopher and Rashidah Haron Galoh methods will catch all expected and unexpected changes to a system but will have problems to detect who did the changes, which is covered in a section further on.

David T. Valys and James P. McGlothlin patent “Automatic reduction of data noise in installation packages for a computer system” [10] where an installation is done on two separate systems followed by calculating a delta can be a way to limit the snapshot drawback mentioned above.

David G. Breggin, Myron Eugene Drapal and Deborah K. Prenger have a patent named “Software installation verification tool” where they describe a way to detect potential problems by comparing an expected state to and actual state of the system. “Briefly, the method and system automatically generates an installation file or database containing information describing or characterizing the installation. A verification tool can compare the installation information to installed information relating to or describing the files actually installed by the install program to locate or identify potential problems” [11]. There method is focusing on verifying that all expected changes have happened but will fail to find additional unexpected changes made to the system.

Richard Brian Singleton patent “Checking computer program installation” [12] describes a similar process where there is a predefined data set that is used to be compared with an actual installation. “The invention provides a specific tool to check the installation of a computer program by examining at least one of the registry entries made or the files stored upon installation. This tool does not seek to reinstall (repair) the computer program, but instead examines the results of an installation that has already been made in comparison with data defining what those results should be for a correct installation and accordingly identifies whether or not a correct installation has taken place.” [12]
3 Theory
3.1 Evaluating different techniques to monitor a file system and registry for changes
All options are evaluated against the following requirements:

1. The implementation needs to be able to track who did the change.
2. The implementation needs to track all changes and cannot afford to lose any.
3. The implementation needs to work on Windows 7 and forward.
4. The implementation needs to support both 32 and 64 bit of Windows.

3.1.1 FileSystemWatcher, ReadDirectoryChangesW and FindFirstChangeNotification API
FileSystemWatcher [13] in .NET platform is using the ReadDirectoryChangesW [14] API call to get information about changes in specific directories. The main limitation with FileSystemWatcher is that it can only monitor one directory at time. Using the ReadDirectoryChangesW API directly would solve that problem.

ReadDirectoryChangesW and its related API FindFirstChangeNotification [15] have a bug that makes it unreliable for small timespans. “If you are using ReadDirectoryChangesW or FindNextChangeNotification to be notified when files are accessed you may not be notified when expected if the file is accessed soon after you were last notified.” [16].

Another problem is related to the internal kernel IO Manager that handles the internal buffered disk write queue. This problem is, that changes are not physically written to the disk before specific conditions are met and hence, these changes will not be detected, as described in the “FileSystemWatcher instability” [17] post on Microsoft “.NET Framework Class Libraries” MSDN forum.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   All the API’s in this section will fail on this requirement since they are only giving information on what has changed and not who did the change.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   This requirement is not met due to bug [16] and [17] mentioned above.

3. **The implementation needs to work on Windows 7 and forward.**
   All API’s are available in Windows XP and forward.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   The API’s are generic and will work on both 32 and 64 bit operating systems.
3.1.2 SHChangeNotifyRegister API

SHChangeNotifyRegister [18] is another API that can be used to monitor file changes in real time. The major drawback with this API is the usage of windows message to deliver the information. This works fine with smaller amount of changes, but when too many changes are happening at the same time we will get a roll-up message just saying “something changed” and hence some of the changes to the system will be lost.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   The API fails on this requirement since they are only giving information on what has changed and not who did the change.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   This API has problems as mentioned above to ensure that all events are collected.

3. **The implementation needs to work on Windows 7 and forward.**
   The API is available in Windows 2000 and forward.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   The API is generic and will work on both 32 and 64 bit operating systems.

3.1.3 NTFS change journal

NTFS file system maintains an “update sequence number” (USN) change journal [19] [20]. For every change made to a specific volume a USN change journal record is created. The main purpose for the change journal is to be used to recover a file system index after for example, computer or volume failure. This solution is something between a real time solution and traditional snapshot solution since no processing happens during the actual execution and no state is saved. This works by saving the USN before the change and after the change, an application can then easily get all changes between those saved USN. This approach does not add any performance or memory overhead during an installation, as a real time solution would do. Also it does not add any waiting time for creating snapshots of a system, as a typical snapshot would do.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   When using NTFS change journal it is not possible to detect who did the change.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   All changes are stored in the NTFS change journal as part of the disk write process.

3. **The implementation needs to work on Windows 7 and forward.**
   NTFS change journal is available in Windows 2000 if using NTFS file system.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   NTFS change journal is implemented and will work on both 32 and 64 bit operating systems.
3.1.4 Snapshot

Snapshot [21] is a technique where the current state of a system is recorded in some form and saved for later comparison. One simple way to do this for file changes are by iterating through all directories on all volumes and gathering all file names. When the same procedure is repeated after an installation all changes can be detected. The more advanced versions do hashing on the file content and compare those results before and after an installation. It is worth to mention that creating a snapshot of bigger file system could easy take a couple of minutes. For registry changes, one simple method is to export the registry data from windows built in regedit application and to use, for example, the free tool regdiff [22].

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   The snapshot technique has problems distinguishing changes made by the process of interest and any other process running in the operating system at the same time. Therefore, it does not meet the requirement.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   Since snapshot is comparing two different states of a system all changes will be found.

3. **The implementation needs to work on Windows 7 and forward.**
   A snapshot can be done on any windows system.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   A snapshot can be done on any windows system.

3.1.5 Kernel file system filter driver (minifilter)

“A file system filter driver intercepts requests targeted at a file system or another file system filter driver. By intercepting the request before it reaches its intended target, the filter driver can extend or replace functionality provided by the original target of the request. Examples of file system filter drivers include anti-virus filters, backup agents, and encryption products.” [23]

A minifilter is a simplified system filter driver where the filter manager takes care of all the complexities of file I/O. This is also the recommended way to implement a new system filter driver by Microsoft. The filter manager moreover, solves one of the big problems with legacy system filter drivers when it comes to controlling the execution order of the applied filters. This is achieved by using an assigned altitude as seen in Figure 1. In this way, it can be ensured that a file filter for an antivirus program is executed before a replication filter, therefore it can detect a virus before the file is replicated. These altitudes are controlled and assigned by Microsoft.
To implement a minifilter we need to create: first, a DriverEntry method that is called when the driver is loaded; then, an unload method that is called during unload of the driver, which is supposed to clean up resources to avoid memory leak and also, close any open user space communication channels. In the DriverEntry method we need to register one pre operation callback per IRP_MJ function code that is of interest for the filter driver. In this callback method any analysis, monitoring, adjustment or even blocking of operation can be done. If the pre operation callback returns an error code, the IRP operation is canceled and no other filters in the chain will be called. It is also possible to register a post operation callback to intercept a specific IRP_MJ function after the operation has been executed. The post operation gives more information about the operation, such as, if it was successful. It is even possible in the post operation to cancel a file operation.

IRP_MJ function codes [24] (I/O request packet major function codes)

- IRP_MJ_CREATE
- IRP_MJ_CREATE_NAMED_PIPE
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_SET_VOLUME_INFORMATION
- IRP_MJ_DIRECTORY_CONTROL
- IRP_MJ_FILE_SYSTEM_CONTROL
- IRP_MJ_NETWORK_QUERY_OPEN
- IRP_MJ_MDL_READ
- IRP_MJ_MDL_READ_COMPLETE
- IRP_MJ_PREPARE_MDL_WRITE
- IRP_MJ_MDL_WRITE_COMPLETE
- IRP_MJ_VOLUME_MOUNT
- IRP_MJ_VOLUME_DISMOUNT

**Requirement evaluation:**

1. **The implementation needs to be able to track who did the change.**
   This is handled by the file manager providing the FltGetRequestorProcessId [25] method which gives access to the process id of the requester.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   Since no change to the file system can bypass the filter and the operating system will wait for the processing to be completed by the system filter, all changes will be recorded.

3. **The implementation needs to work on Windows 7 and forward.**
   No information has been found but it has been tested on windows 7 and windows 8

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   To meet the 32/64 bit requirement it is only needed to recompile the code with different target architecture.

3.1.6 **API hooking**

Hooking [26] is a technique to alter the behavior of an application or operating system, by intercepting function calls or message events that are sent between different parts of a software implementation. Since we are interested in file changes we could hook the kernel32.dll and have CreateFileW [27] function redirected to our own function, where we log this call and return the call back to CreateFileW to finalize the call as seen in Figure 2. This is also referred to as kernel patching. “Kernel patching is the practice of using internal system calls and other unsupported mechanisms to modify or replace code or critical structures in the kernel of the Microsoft Windows operating system with unknown code or data. "Unknown code or data" is any code or data that is not provided by Microsoft as part of the Windows kernel.” [28]. Kernel patching works fine on all x86 version of windows even if Microsoft does not recommend doing this type of modification to the system because it reduces considerably system security and reliability.

For every new version or update of Windows, some part of the API may change. Therefore, a hook that worked fine in a previous version might not work anymore and can result in unpredictable behavior, system instability and “Blue Screen of Death” [29]. With the introduction of 64 bit versions of Windows, Microsoft chose to implement a technical barrier to hooking called Kernel Patch Protection [28]. “Kernel Patch Protection monitors if key resources used by the kernel or kernel code itself has been modified. If the operating system detects an unauthorized patch of certain data structures or code it will initiate a shutdown of the system.” [30].
The first thing when applying a hook is to load the hook code into the context of the application that should be hooked, by injecting a hook dll containing a set of hook functions with the same signature as the target function. Accomplishing the redirect to the hook function can be done by:

- Altering the SSDT (System Service Dispatch Table) that contains pointers to functions in ntoskrnl.exe.
- Altering the IAT (Import Address Table) that contains pointers to imported functions.
- Altering the VMT (Virtual Method Table) that contains pointers to virtual methods. A virtual method is in many programming languages a method that can be overridden by inheriting classes and the compiler will add an entry for it in the VMT.
- And more.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   Since the hook is executed in the same context as the calling method the process id will be the same, hence we know who did the change.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   Since the hook is placed in the execution chain of the operation as seen in Figure 2, this solution will catch all changes.

3. **The implementation needs to work on Windows 7 and forward.**
   API hooking works on any version of windows.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   Since API hooking is limited to 32 bits system due to “Kernel Patch Protection” it does not meet this requirement.
3.1.7 Event tracing for windows

“Event Tracing for Windows (ETW) is an efficient kernel-level tracing facility that lets you log kernel or application-defined events to a log file. You can consume the events in real time or from a log file and use them to debug an application or to determine where performance issues are occurring in the application. ETW lets you enable or disable event tracing dynamically, allowing you to perform detailed tracing in a production environment without requiring computer or application restarts.” [31]

![Event tracing architecture](image)

Controllers as seen in Figure 3 take care of managing the ETW sessions. They can start/stop ETW sessions, enable providers to log events, manage the size of the buffer and obtain statistics for a session. To create a session or enable a provider, the controller needs to run with administrative privileges.

Providers as seen in Figure 3 can be both an application containing event tracing instrumentation or a NT kernel logger providing system events from the operating system, for example, disk IO or page fault events.

Consumers as seen in Figure 3 are applications which use ETW sessions created by the controller to consume the events provided by the session in a chronological order. ETW sessions created by the controller to consume the events provided by the session in a chronicle order.

When consuming events, there is a risk that some events will be missed. This can happen if:

- The total size of an event exceeds the size of 64kB (the default buffer size), then the event cannot transferred and will be lost. The total size of an event includes both the payload and the ETW header.
- The total size of the event exceeds the size of the EWT buffer, then it cannot be saved and hence will be lost.
- An application does not consume events fast enough when the processing is in real time, then the backing file will fill up and no more events can be stored in the backing file. This will lead to all events happening during the time the application is working on catching up, will be lost.
- When directing the logging to a disk that is slow, the events will be lost.

To reduce chances of losing any events, it is possible to modify some parameters when creating ETW sessions. Bellow parameters have been identified by James Rapp to be the important ones to adjust, to avoid losing events [32]. All possible parameters can be found in the EVENT_TRACE_PROPERTIES structure [33].

- **FlushTimer**
  Flush timer defines how often in seconds the buffer should be flushed forcibly. This flush is an additional one to the automatic ones that happen when a buffer is full. James Rapp recommends a value of 0 which means that the buffer will be flushed once every second.

- **BufferSize**
  The maximum size of the buffer is 1Mb. In general, this size needs to be big if the event rate is expected to be high. Even when the expected event rate is low, it is recommended not to have too small buffer since it will increase the rate of flushing. James Rapp recommends a value of 256 kB as the size.

- **MinBuffer**
  Minimum number of buffers. James Rapp recommends a value of 512.

- **MaxBuffer**
  Maximum number of buffers. James Rapp recommends a value of 1024.

Microsoft provides a NuGet packet which is a managed library to interact with the ETW called “Microsoft TraceEvent Library” [34]. This library currently has a bug when it comes to providing process information for FileIO/FileCreate and FileIO/FileDelete events, and hence, it is not possible to know who created or deleted a file.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   ETW does provide a process id for all changes except for FileIO/FileCreate and FileIO/FileDelete events due to above mentioned bug, hence the requirement is not met.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   All changes will be captured as long as the buffer does not get full. The risk of this happening is very small, as James Rapp mentions, but it is still theoretically possible to miss an event, hence this requirement is not met.

3. **The implementation needs to work on Windows 7 and forward.**
   ETW works on windows Vista and onward.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   ETW is not limited to operating system version
3.1.8 Kernel registry filter driver

Microsoft describes registry filter like: “A registry filtering driver is any kernel-mode driver that filters registry calls, such as the driver component of an antivirus software package. The configuration manager, which implements the registry, allows registry filtering drivers to filter any thread's calls to registry functions” [35].

This method is used by antivirus software to ensure that viruses do not get access to specific registry keys. It is also possible to monitor registry calls or modify a registry call with a registry filter driver.

To implement a registry filter driver, a callback routine needs to be registered by using the CmRegisterCallback [36] method as part of the registry filter driver DriverEntry method. After a successful registration, all changes to the registry will be routed to this registered callback method and, no registry change can happen before the callback routine exits. Hence, the system performance can be heavily impacted if not carefully implemented because windows operating system can easily generate thousands of changes per seconds.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   Since the callback routine registered with CmRegisterCallback [36] is running in the same process as the calling process doing the registry change, PsGetCurrentProcessId [37] method can be used to find what process is doing the change.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   Since all registry calls will be routed to the registry filter, the requirement about not losing any event is ensured.

3. **The implementation needs to work on Windows 7 and forward.**
   Register filter is implemented in windows XP and onward.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   To meet the 32/64 bit requirement it is only needed to recompile the code with different target architecture.
3.1.9 System Registry provider in WMI

“The System Registry provider creates several methods and event classes that allow Windows Management Instrumentation (WMI) scripts or applications to interact with the registry. You can subscribe to several registry events. For example, you can subscribe to the changes in a registry key, in a sub key, or in a registry key value.” [38]. The strength by using WMI is that you can subscribe to both local and remote events on other hosts which opens up for monitoring multiple machines during a complex installation. The major problem is the limitations on what hives can be monitored ("HKEY_CLASSES_ROOT" and "HKEY_CURRENT_USER" are the hives that cannot be monitored [38]). Another problem is that the information provided by the events do not contain any information about the values, change type or who did the change. The information provided is: where and when the change happened.

The provided events:

- RegistryTreeChangeEvent [39] is an event representing changes to a key or its sub keys.
- RegistryKeyChangeEvent [40] is an event representing changes to a specific key and not to its subkeys.
- RegistryValueChangeEvent [41] is an event representing a change to a single value of a specific key.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   No information about who did the change is provided as part of the event, hence this requirement is not met.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   Since "HKEY_CLASSES_ROOT" and "HKEY_CURRENT_USER" cannot be monitored [38] all changes will not be recorded.

3. **The implementation needs to work on Windows 7 and forward.**
   System registry provider is implemented in windows Vista and onward.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   System registry provider is a generic implementation not limited by windows versions.
3.1.10 Process Monitor

“Process Monitor is an advanced monitoring tool for Windows that shows real-time file system, Registry and process/thread activity. It combines the features of two legacy Sysinternals utilities, Filemon and Regmon, and adds an extensive list of enhancements including rich and non-destructive filtering, comprehensive event properties such session IDs and user names, reliable process information, full thread stacks with integrated symbol support for each operation, simultaneous logging to a file, and much more. Its uniquely powerful features will make Process Monitor a core utility in your system troubleshooting and malware hunting toolkit.” [42]

It is possible to automate the usage of process monitor with command line arguments.

- To start process monitor run “procmon /quiet /minimized /backingfile logFile.pml”.
- To stop process monitor run “procmon /terminate”
- To export the log in xml format run “procmon /openLog logFile.pml /SaveAs logFile.xml”

Process monitor log is very verbose. Performing an actual test with a simple installer (default settings during 3 seconds) captured over 150.00 events and generated a log file of over 70Mb. This gives around 1.2-1.5 GB of data per minute and, on a long running installation session of many minutes, the log file can be of essential size.

Requirement evaluation:

1. **The implementation needs to be able to track who did the change.**
   All events from process monitor contain information about who did the change.

2. **The implementation needs to track all changes and cannot afford to lose any.**
   All events are recorded

3. **The implementation needs to work on Windows 7 and forward.**
   Process monitor runs on windows XP SP2 and higher.

4. **The implementation needs to support both 32 and 64 bit of Windows.**
   Process monitor has one version for 32 bit and one for 64 bit.
3.1.11 Selecting the way to monitor the system

Requirements:

1. The implementation needs to be able to track who did the change.
2. The implementation needs to track all changes and cannot afford to lose any.
3. The implementation needs to work on Windows 7 and forward.
4. The implementation needs to support both 32 and 64 bit of Windows.

<table>
<thead>
<tr>
<th>Option</th>
<th>Req 1</th>
<th>Req 2</th>
<th>Req 4</th>
<th>Req 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileSystemWatcher, ReadDirectoryChangesW and FindFirstChangeNotification API</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SHChangeNotifyRegister API</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NTFS change journal</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Snapshot</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Kernel file system filter driver (minifilter)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>API hooking</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Event tracing for windows</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Kernel registry filter driver</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>System Registry provider in WMI</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Process Monitor</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 1: All options evaluated against the requirements

From Table 1 the techniques that meet the requirements are:

- kernel filter system driver
- kernel registry driver
- process monitor

Due to the delimitation of this thesis being a solution based on the .NET platform, the first two techniques are ruled out. Process monitor is the selected way and is giving a good balance between data quality, execution speed and implementation time.
3.2 Evaluation of methods to filter event data

Recording all changes happening in the system during an installation can generate more than 50,000 events per second (measured during multiple installations on a test system with a minimum number of other processes and services running). Since an installation can take multiple minutes in some cases, the total set of recorded events can easily exceed 30 million events. To find the events that are needed to verify if an installer has done its work correctly, we need a way to filter out those events.

To test the efficiency of all filters, a set of different event collections have been used. This has been captured from multiple installations executed on a test system with a minimum number of other processes and services running. The captured data set ranges from 3 seconds to 3 minutes.

3.2.1 Filter on processes related to the installation

A process is an instance containing all the code and its current state of a running computer program. Every time a user starts an installation of a windows installer package, a process named “msiexe.exe” is started to handle the installation. By searching for the start event of this specific executable, that has our installation package as an argument in all event records, we can extract the process ID for the windows installer process of interest. When the main process ID has been found a search for all processes started by the main process is conducted in a recursive way, to collect all child processes and their child processes as seen in Figure 4. Since windows installer can delegate some work to an already executing windows installer process, we also need to find all “msiexe.exe” and their child processes that were running in the time frame of our installation. When this is done, all recorded events can be filtered based on all process ID that are of interest. Applying this filter on the specified test data leaves an average of around 10% of all events.

```
<event>
  <ProcessIndex>19709</ProcessIndex>
  <Process_Name>msiexec.exe</Process_Name>
  <PID>5800</PID>
  <Operation>Process Start</Operation>
  <Path></Path>
  <Result>SUCCESS</Result>
  <Detail>Parent PID: 2368, Command line: &quot;C:\Windows\System32\msiexec.exe&quot; /qn /x &quot;Installers\Installer2.msi&quot; /l*v &quot;C:\Temp\tmpB1A3.tmp&quot; 
...
</Detail>
</event>
```

*Figure 4: Process start event*

Other none windows installer based installations would use the same approach.
3.2.2 Filter event based on event type
Since it is only a limited amount of event types that is of interest, an approach would be to remove all other events.

Events of interest

- **Create file**
  This event can both indicate that an application is accessing a file or if a file is actually created. A create file event is created for example when the CreateFileW [27] method is executed.

- **Set dispositions information file**
  This event can indicate if a file has been deleted.

- **Set rename information file**
  This event is important since windows installer first renames the file before deleting it.

- **Process start**
  This event is needed to indicate when the installer process starts. This can be used when doing process filtering.

- **Process exit**
  This event is needed to indicate when the installer process stops. This can be used when doing process filtering.

- **Registry set value**
- **Registry delete value**
- **Registry create key**
- **Registry delete key**

Filtering the test data set by above event types and on only events that were successfully executed (an unsuccessful event is not of interest since nothing changed), leaves an average of around 10% of all events.

3.2.3 Filter events based on windows installer internal housekeeping
Windows installer does a lot of internal housekeeping during execution to enable, for example, uninstallation. All of those events are not unique for our installer and the order can change between different versions of windows installer. Removing all windows installer internal housekeeping does reduce the number of events on average by half when applied on the test data.

Below is a list of registry key locations that are used by windows installer for internal housekeeping

- **HKLM\Software\Microsoft\Windows\CurrentVersion\Installer\** and the 64 bit version **HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Installer\** is used to store data about what product, patches, components and much more have been installed and where.

- **HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall\** and the 64 bit version **HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall\** is used to store information for uninstallation [43]. When executing “Add remove programs” on windows 8, this registry key is used to populate the table in the program with information about all installed programs and their publishers, among many things. See full list of field’s reference [43]

- **HKLM\Software\Classes\Installer\** is used to store data on what has been installed on a per machine bases. [44]
- `HKCU\Software\Microsoft\RestartManager\` is used to store information for the restart manager. [45]
- `HKCU\Software\Microsoft\Installer\` is used to store data on what has been installed on a per user basis. [44]
- `HKCR\Installer\`

Below is a list of directories used by windows installer for internal housekeeping:
- `C:\Windows\Temp\` is used for temporary files by windows and any application can add or remove files from there whenever they want. It is normally used for temporary runtime files.
- `C:\Windows\Installer\` is as caching directory to enable repair without the original source being present. [46]
- User own temp directory. On Windows 8 it is located in `C:\Users\[User Name]\AppData\Local\Temp`

3.2.4 Filter events based on timespan
If we know when our installation process starts and stops, it is possible to remove all events outside this timespan. This will not add much value if we are also filtering events based on processes related to installation.

3.2.5 Selecting the way to filter events
A combination of all filters, except of timespan filter, gives the best result as can be seen in Figure 5 bellow.

![Flow diagram for filtering events](image_url)
4 Implementation

4.1 System overview
The system consists of five major areas that, in conjunction, create a complete solution for automating installation testing. Each area consists of a group of classes, libraries and methods which have been grouped as follows:

- **Installation Manager** gives a way to control and interact with the software installer being tested.
- **Installation Verifier** is responsible for verifying that only the expected changes have happened and no unexpected ones occur.
- **System Data Analyzer** is responsible for interpreting the raw system data coming from the System Monitor. This area uses the filter methods studied in section 3.2.
- **Test Data Loader** is responsible for loading the correct test data used by the installer verifier.
- **System Monitor** is responsible for collecting all changes on the system where the software installer is being tested on. This area uses the selected way studied in section 3.1.

4.1.1 System workflow
1. Load test data and verify its correctness (Test Data Loader)
2. Start System Monitor
3. Execute the installer in the selected mode (Installation/Uninstallation/Repair)(Installation Manager)
4. Stop System Monitor
5. Analyze the raw system event data and extract all interesting file events and registry events (System Data Analyzer)
6. Execute the Installation Verifier
7. Evaluate result

4.2 Installation Manager
The Installation Manager is responsible for interacting and controlling the execution of the installer. This is done by using the .Net Process [47] class.

The Installation Manager, when working with a windows installer package, supports the possibility to define:

- A list of features to be installed: A windows installer package feature [48] is a list of child features and/or a list of components [49] that is a piece of the product to be installed. “Examples of components include single files, a group of related files, COM objects, registration, registry keys, shortcuts, resources, libraries grouped into a directory, or shared pieces of code such as MFC or DAO.” [49].
- A list of public properties: public properties [50] are normally used by windows installer packages to change default behaviors as installation location or installation level [51].
4.3 Installation Verifier

The installation verifier is looping through a list of test data items to ensure that the system has been changed accordingly. This is done by splitting the verification into smaller specialized checks. Each check is responsible to check its own test data items and to report the result back. Every check also removes the files and/or registry changes that they have verified. This is made to avoid other checks using the same changes. After all checks have been executed, it is verified that all file and/or registry changes reported by system monitor have been accounted for.

As seen in Figure 6 a check consists of:

- a Check class that is responsible for doing the actual verification
- a corresponding TestData class that is a container class with all data needed by the Check class
- a Loader class responsible for loading test data items from a test data definition file

The solution contains the following built in checks:

- File check verifies the existence of a file and its version.
- Directory check verifies the existence of a directory.
- Registry key check verifies the existence of a specific registry key.
- Registry value check verifies the existence of a specific registry value and its content.
- Start menu check verifies the existence of a specific start menu.

It is also possible to extend the system with external checks see in section 4.10.
4.4 System Monitor

System Monitor is responsible for monitoring the system where the software installer is being tested and for capturing all change events of interest. System monitor does filter out all events that are needed by the System Data Analyzer seen in Figure 7. System Monitor has three methods:

- Start()
- Stop()
- GetEventData() : Returns a collection of all collected events.

When monitoring the system, only the events that are needed in further processing are saved.

![Diagram](image)

*Figure 7: Filter based on event type*

4.5 System Data Analyzer

System Data Analyzer goes through all events collected by System Monitor and creates one list of all file changes of interest and one list of all registry changes of interest. Before file and registry changes can be extracted, a process filter is applied as shown in Figure 8. This is done by finding the start process and all its child processes recursively followed by filtering all events based on the process found of interest. After all registry and file events have been found, a second filter is applied to remove all windows installer internal housekeeping noise, as seen in Figure 8.

![Diagram](image)

*Figure 8: Filter based on related process*
4.5.1 Find file changes

To find what file changes have happened, we first need to filter out events not related to the installation process we are looking at as seen in Figure 9. It is essential that all events are analyzed in time order to avoid finding, for example, that a file is delete before it has been created. This would lead to a lot of noise and an unpredictable solution. The file changes extractor is designed to take into account the following windows installer behaviors:

- **Windows installer uninstallation behavior**
  1. Rename file to temporary name
  2. Delete the temporary file

- **Windows installer repair behavior**
  1. Rename file to temporary name
  2. Create the new file
  3. Delete the temporary file

![Figure 9: Find files changes workflow](image-url)
4.5.2 Find registry changes

To find what registry changes have happened, we first need to filter out events not related to the installation process we are looking at, as seen in Figure 10. It is essential that all events are analyzed in time order to avoid finding, for example, a registry deleted before it has been created. This, again, would lead to a lot of noise and an unpredictable solution.

![Diagram](image_url)

Figure 10: Find registry changes workflow
4.6 Test Data Loader

Test Data Loader is used to load test data from xml files. The flow to load test data:

1. Resolve all include statements in the file. This is mostly used for sharing common data, such as, properties between different setups. It can be compared to “using” in C# or “#include” in C++.
2. Load all referred files recursively and merge, if they exist, properties, msifiles, components, plugin elements in both parent and referred file.
3. Extract all properties and resolve its content. See more under properties for explanation of resolve.
4. Extract all msifiles that are used when loading MSI feature from other test files.
5. Extract all plugins that are used to load additional checks.
6. Load all features by using specific test data loader per check.
7. Collect expected test data from all features and return result.

4.6.1 Test data structure for install chainer files

The install chainer test data definition file does contain all test data needed to verify the result after a test run.

From Figure 11, we can see that the installer chainer element can contain include, components, msifiles, properties and plugins. The “include” element is used to include additional test data from other test data files.

As seen in Figure 11, a component can contain one or more msifeatures. Every component element needs to have a unique id attribute used in error logging to identify the problem area.

The msifeature element has two configuration attributes. The first one, named “msiID”, is a unique identifier for the MSI where the “featureName”, the second attribute, is defined. The “msiID” in the msifeature element is used in conjunction with a misfile element to load a specific MSI feature from another test data definition file. For example the msifeature on line 7 in Figure 11 will load a feature named “Feature2” from “msi2.xml” that is referred in the misfile element on line 15.

Properties element is covered in section 4.6.3 in great details and the plugin element is covered in 4.6.2.
<?xml version="1.0"?>
<installChainer>
  <include ref="CommonConfiguration.xml"/>
  <components>
    <component id="comp1">
      <msifeature name="Feature1" msiID="msi1"/>
      <msifeature name="Feature2" msiID="msi2"/>
    </component>
    <component id="comp2">
      <msifeature name="Feature1" msiID="msi1"/>
    </component>
  </components>
  <msifiles>
    <msifile path="msi1.xml" id="msi1"/>
    <msifile path="msi2.xml" id="msi2"/>
  </msifiles>
  <properties>
    <property type="Text" id="TmpFilePath" value="C:\tmp"/>
    <property type="RegistryValue" id="DirectXVersion"
      key="HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectX"
      valueName="Version" valueIfNotFound="1"/>
    <property type="WindowsInstaller" id="ComputerName"/>
  </properties>
  <plugins>
    <plugin path="ExampleExtension.dll" datatype="File2"/>
  </plugins>
</installChainer>

Figure 11: Install chainer test data file
4.6.2 Test data structure for MSI files

The MSI test data definition file does contain all data needed to verify the result after a test run. In Figure 12 we can see that the MSI element can contain include, features, msifiles, properties and plugins. Properties element is covered in section 4.6.3 and msifiles is covered in section 4.6.1. A feature does contain one or more verification test data items, that is, the smallest verification part used by checker classes to verify a single statement. For example, in Figure 12 on line 7, the test data item defines that a file named “payload1.txt” should be verified. For more details about verification test data see section 0.

When the built in verification test data does not fulfill a specific testing need, a separate plugin can be created to handle this new requirements as described in section 4.10. In Figure 12, line 25, there is a plugin named “ExampleExtension.dll” containing a FileSize verification pair used in line 8 in the same figure.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<msi id="msi3">
  <include ref="CommonProperties.xml" />
  <features>
    <feature name="Default">
      <directory path="[FOLDER1]" />
      <file path="[FOLDER1]\Payload1.txt" />
      <fileSize path="[FOLDER1]\Payload3.txt" size="15478" />
      <registrykey key="HKCU\Software\MySoftware\Installer1" />
      <registryvalue key="HKCU\Software\MySoftware\Installer1" name="(Default)" data="He there" />
    </feature>
    <feature name="Client">
      <startmenu allusers="true" name="New Client"/>
      <msifeature name="Feature2" msiID="msi2"/>
    </feature>
  </features>
  <properties>
    <property id="FOLDER1" type="Parameter" parameterID="folder1" />
  </properties>
  <msifiles>
    <msifile path="msi2.xml" id="msi2"/>
  </msifiles>
  <plugins>
    <plugin path="ExampleExtension.dll" datatype="FileSize"/>
  </plugins>
</msi>
```

*Figure 12: MSI test data file*
4.6.3 Properties in test data files

Properties entry is used to only define common used values in a single place in the system. A property can be on MSI test data xml context, install chainer context or system global context. Examples of all property types can be seen in Figure 13.

Property attributes:

- **type (Mandatory)**
  The type of property
  - **Text**
    A simple string property
  - **RegistryPathExist**
    Will return a string “True” or “False” depending if the registry path exists
  - **RegistryValue**
    Will search the registry for a specific value and add that as a property
  - **WindowsInstaller**
    Will search for a windows installer property [52]
  - **SpecialFolderPath**
    Will use the built in .NET environment class to get a value of special folders, for example, “ProgramFilesFolder”
  - **EnvironmentVariable**
    Will take a value from the environment and add it as a property
  - **Parameter**
    Is used to link a parameter in the setup configuration for a MSI or an install chainer

- **Id (Mandatory)**
  The name of the property

- **value**
  The value (it may contain other properties), used by type EnvironmentVariable and Text

- **valueIfNotFound**
  Used by type WindowsInstaller and RegistryValue to set a default value if the property was not found

- **key**
  Used by type RegistryPathExist and RegistryValue to give key path into the registry

- **view**
  Used by type RegistryValue to define if 64 bit or 32 bit scope should be used of the registry

- **valueName**
  Used by type RegistryValue to define what value to get from the key

- **parameterID**
  Used by type Parameter to identify what parameter the property is linked to
```xml
<?xml version="1.0" encoding="utf-8" ?>
<properties>
  <property type="SpecialFolderPath" id="PFFolder" value="ProgramFiles"/>
  <property type="EnvironmentVariable" id="Windows" value="windir" />
  <property type="Text" id="TmpFilePath" value="C:\tmp"/>
  <property type="RegistryValue" id="DirectXVersion"
    key="HKLM\SOFTWARE\Microsoft\DirectX"
    valueName="Version" valueIfExists="1"/>
  <property type="WindowsInstaller" id="VersionNT64" valueIfExists="0"/>
  <property type="RegistryPathExist" id="UnInstallRegistry"
    key="HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall"/>
  <property type="Parameter" id="Property1" parameterID="PROPERTY1"/>
</properties>
```

*Figure 13: Properties test data file*
4.7 Verification test data
4.7.1 Common options for all test data items
- NeverUnInstall (optional)
  It is used to indicate that the test data item will not get uninstalled after it is installed. In the example below, the database file that cannot be removed during uninstallation because it could contain customer data.
  
  ```xml
  <file path="[VirtualDirectories]\[WEBAPPPORT]\web.bak" neverUnInstall="true" />
  ```
- Condition (optional)
  Condition is used in cases where test data items may only appear on a system under specific conditions, for example, a specific version of Windows. This condition is normally used to mirror the condition attribute on components in a windows installer package file. In the example below, the file will only get installed if the Windows version is bigger than 600 for a 32 or 64 bit OS version. Windows versions 600 is the Vista release [53]. Properties “VersionNT” and “VersionNT64” need to be set directly in the xml file or in global properties file for the bellow example to work.
  
  ```xml
  <file path="[INSTALLDIR]\Microsoft.Dynamics.Nav.ManagementUI.dll" condition="([VersionNT] &gt; 600) OR ([VersionNT64] &gt; 600)" />
  ```

4.7.2 File test data
The file check is using the file test data to verify the existence or not of a file. The file version can be verified as an additional and optional check.

Attributes:
- path : full path to the file (mandatory)
- file version : used to verify the file has the correct file version

Examples:
  
  ```xml
  <file path="[ProgramFiles]\Windows Media Player\mplayer2.exe" />
  <file path="[INSTALLATIONDIR]\File3.txt" fileVersion="1.5.0.0" />
  ```

4.7.3 Start menu test data
The start menu check is using the start menu test data to verify the existence or not of a start menu.

Attributes:
- name : the name of the start menu (mandatory)
- allusers : defines if the start menu item is installed for all users or only for the one executing the installer

Examples:
  
  ```xml
  <startmenu allusers="true" name="New Client"/>
  ```
4.7.4 Registry key test data
The registry key check is using the registry key test data to verify the existence or not of a specific registry key. See Figure 14 for an example of a registry key.

![Registry Editor screenshot](image.png)

Figure 14: Screenshot of regedit showing registry key structure

Attributes:

- **key**: complete key (mandatory)

Example:

```xml
<registryKey key="HKCU\Software\Installer1" />
```

4.7.5 Directory test data
The directory check is using the directory test data to verify the existence or not of a directory.

Attributes:

- **path**: Full path to the file (mandatory)

Example:

```xml
<directory path="[INSTALLDIR]" />
```
4.7.6 Registry value test data

The registry value check is using the registry value test data to verify the existence or not of a registry value and its content. See Figure 15 for an example of a registry value named “SomeIntegerValue” with data value of “1”.

![Registry Editor Screenshot](image)

Figure 15: Screenshot of regedit showing an integer value

Attributes:

- key: complete key (mandatory)
- name: the name of the value to check (mandatory)
- data: the value
- dataCmpMethod: defines in what way we should compare the data. The values can be either “Equal” or “Contains”. The default method is “Equal” if no other method has been defined

The following example verifies both values in Figure 15. The first one is using equal, the default value as dataCmpMethod is missing. The second one is using contains, matching the partial value seen in Figure 15.

```xml
<registryValue key="HKCU\Software\Installer1" name="SomeIntegerValue" data="1" />
<registryValue key="HKCU\Software\Installer1" name="(Default)" data="There" dataCmpMethod="Contains" />
```
4.8 Logging

A very important part of the verification is the ability to log the result of the execution. Without this, it would be impossible to interpret what did fail or what did get checked.

Different listeners can be added to the built in logger. A listener is a class that writes the logging data to different media, as for example the event log. There are six different built in listeners:

- **Console log listener**
  The log is written directly to the standard output

- **Debug log listener**
  The log is written to the debug console

- **Event log listener**
  The log is written to windows system event log

- **Stream log listener**
  The log is written to a stream of any type

- **Text file log listener**
  The log is written to a text file

- **Trace log listener**
  The log is written to the trace console

They way to create a new listener is to create a class that inherits from the BaseLogListner class. As part of the implementation, a method called Log needs to be overridden as seen in Figure 16. The Log method accepts an object of type LogData.

LogData properties:

- **Message**
  The actual message

- **ErrorType**
  Can be Error, Warning or Information

- **Time**
  When the event happened

- **Where**
  Where the event happened

- **Indent**
  The indent
namespace InstallerVerificationLibrary.Logging
{
    using System.Diagnostics;

    public class EventLogListener : BaseLogListener
    {
        private readonly string log = "Application";
        private readonly string source = "Installer Testing Library";

        public EventLogListener() : this(LogErrorType.Information)
        {
        }

        public EventLogListener(LogErrorType type) : base(type)
        {
            if (!EventLog.SourceExists(source))
            {
                EventLog.CreateEventSource(source, log);
            }
        }

        public override void Log(LogData data)
        {
            var message = data.Where + " -> " + data.Message;
            if (data.ErrorType == LogErrorType.Error)
            {
                EventLog.WriteEntry(source, message, EventLogEntryType.Warning);
            }
            if (data.ErrorType == LogErrorType.Information)
            {
                EventLog.WriteEntry(source, message, EventLogEntryType.Information);
            }
        }
    }
}

Figure 16: An example log listener

The logging process flow is:

1. Create an instance of a listener with the preferred logging level, defined as: error, warning or information. When registered for error, only error messages will be sent to the added listener. When registered for warning, both errors and warnings will be sent to the listener. Information sends errors, warnings and information to the listener.
2. Call StartLogging() on the new created listener instance.
3. When something needs to get logged, one of the three main methods on the static Log class is called
   a. WriteError
   b. WriteWarning
   c. WriteInformation
4. A LogData item is created and dispatched to all listeners registered for specified error level
5. The listener decides how the information should be logged
4.9 Tools
Tools are a group of common tools/classes that are used by different parts of the solution. These tools can be used directly in tests or extensions.

4.9.1 XML tool
The XML tool class contains methods to work with XML files. This tool is heavily used in the test data loader classes as can be seen in Figure 18 in section 4.10.2

Methods:
- **GetNamedAttributeValue**
  Returns an attribute value if exist or a default value.
- **GetOuterXml**
  Returns the outer xml string of an xml element.

4.9.2 Collection tools
The Collection tool class contains different methods to manipulate a generic collection of objects.

Methods:
- **CopyCollection**
  Returns a new collection with a copy of all objects from the parent collection.
- **CloneCollection**
  Returns a new collection with a clone of all objects from the parent collection. This is useful to ensure that changes to a reference object are not reflected in the other collection.
- **MergeCollection**
  Merge two collections.

4.9.3 Application tool
The Application tool class contains methods for install/uninstall/repair of MSI and execution of exe files.

Methods:
- **InstallMSI**
  Installs an MSI file and writes the log output form msiexec.exe to a log file
- **RepairMSI**
  Repairs an MSI file and writes the log output form msiexec.exe to a log file
- **RepairMSIByProductCode**
  Repairs an MSI file based on the product code
- **UninstallMSI**
  Uninstall an MSI file and writes the log output form msiexec.exe to a log file
- **UninstallMSIByProductCode**
  Uninstall an MSI file based on the product code
- **RunExe**
  Runs an exe file with the provided parameters and waits for the process to exit or timeout.
4.9.4 Database tool
The Database tool is a class containing methods to work against a MS SQL database server. If a software installer handles a database, these set of methods can be used in different scenarios. For example, by adding a database before the installer is executed to simulate an already existing database.

Methods:

- **DropDatabase**
  Drops a database on a specific server and instance

- **DatabaseExist**
  Returns true if a specific database exists

- **GetDatabases**
  Returns a collections of database names on a specific server and instance

- **UserIsMemberOfRole**
  Returns true if the defined user is member of the defined database role

- **AttachDatabase**
  Attaches a database file on specific server and instance

4.9.5 File system tool
The File system tool class contains methods related to the file system. For example, it has a method that will move around some bits in a binary file to be used in a repair test scenario.

Methods:

- **RemoveFile**
  Removes a specific file from the file system and logs the action

- **RemoveDirectory**
  Removes a specific directory from the file system and logs the action

- **FileCopy**
  Copies a file

- **DestroyTextFile**
  Moves around text parts in a text file to simulate that the file have been corrupted

- **FileExist**
  Returns true if a specific file exists

- **CheckIfTextFileIsRestored**
  Checks if a specific file contains a specified string

- **DestroyBinaryFile**
  Moves around bytes in a binary file to simulate that the file has been corrupted
4.9.6 Registry tool
The Registry tool class contains methods to work with the registry. The tool does implement an improved way
to handle 32 and 64 bit registry keys. By defining a view property it is possible to access either the 32 bit or 64
bit version of the registry key, independent if the calling process is 32 or 64 bit. The .Net implementations only
allow a 64 bit process to access the 64 bit view of the registry. Same applies to a 32 bit process.

Methods:

- **RemoveRegistryKey**
  Removes a specific registry key and its sub keys.

- **RegistryKeyExist**
  Returns true if a specific registry key exists.

- **RegistryValue**
  Returns the data for a specific named key value pair or a default value.

4.9.7 Windows service tool
The Windows service class contains methods to remove services. This can be used, for example, when testing a
repair scenario of a software installer that creates a windows service.

Methods:

- **RemoveService**
  Removes a single service from the system

- **RemoveServices**
  Removes multiple services passed on a name search pattern or based on a list of names

4.9.8 Fuzz tool
The Fuzz class is a set of methods that can be used for fuzz testing [54]. Fuzz testing is a testing technique
where a system is tested by giving invalid, unexpected or random data as input.

Methods:

- **RandomNumber**
  Returns a random integer between a minimum value and maximum value.

- **UnicodeChar**
  Returns a single random Unicode character.

- **AsciiChar**
  Return a single random ASCII character.

- **UnicodeString**
  Returns a string with a specific length filled with random Unicode characters.

- **AsciiString**
  Returns a string with a specific length filled with random ASCII characters.
4.9.9 Manual Uninstaller tool

The Manual uninstaller class is used to ensure that any trace of an installation is removed from the machine. This is normally used as a cleanup step after each test execution to ensure we have a known state of the system. This is to avoid test failing for random reasons due to unexpected state of the system.

The different areas that the tool can clean

- Databases
- Directories
- Processes
- ProductCodes
- Services
- RegistryKeys
4.10 Adding additional checks by using the plugin feature

To create a new specific check to be used by the solution, 3 classes are needed:

- Check class
- Loader class
- TestData class

To enable the solution to load the new extension, a plugin element needs to be added to the test data definition file, see section 4.6.2

4.10.1 Check class

All check classes need to inherit from the “BaseCheck” class as seen in Figure 17. Additionally, the classes need to have two attributes. The first attribute is the “TestCheck” defining that this class is a test check class. The second one is the “DataType” attribute, defining what test data this check class consumes.

```csharp
namespace InstallerVerificationLibrary.Check
{
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using InstallerVerificationLibrary.Attribute;
    using InstallerVerificationLibrary.Data;

    [TestCheck]
    [DataType("FileSize")]
    public sealed class FileSizeCheck : BaseCheck {
        public override CheckResult DoCheck(BaseTestData data, ICollection<FileChange> fileChanges, ICollection<RegistryChange> registryChanges) {
            if (!VerifyIfCorrectTestData(data)) {
                return CheckResult.NotCheckDone();
            }

            var fileData = data as FileSizeData;
            var fileChange = fileChanges.FirstOrDefault(x => x.Path == fileData.Path && x.IsDirectory == false);

            var fileExistenseResult = VerifyFileExistense(fileChanges, fileChange, fileData);
            if (!fileExistenseResult.Success) {
                return fileExistenseResult;
            }

            // Check file size
            var info = new FileInfo(fileData.Path);
            if (info.Exists) {
                if (fileData.Size != info.Length) {
                    fileChanges.Remove(fileChange);
                    return CheckResult.Failure("File=" + fileData.Path + " does have wrong file size. Found size:" + info.Length + " expected size:" + fileData.Size + ",");
                }
            }

            fileChanges.Remove(fileChange);
            return CheckResult.Succeeded(data);
        }
    }
}
```

*Figure 17: Check class example*
4.10.2 Loader Class

The loader class reads data from xml test data definition files. The loader class needs to be marked with the same “DataType” attribute as the check class, and also, be marked with the “TestDataLoader” attribute as seen in Figure 18. The “TestDataLoader” attribute is used by the solution to define what loader to use for different xml elements.

```csharp
namespace InstallerVerificationLibrary.Loader
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Linq;
    using System.Xml.Linq;
    using InstallerVerificationLibrary.Attribute;
    using InstallerVerificationLibrary.Data;

    [DataType("FileSize")]
    [TestDataLoader]
    public sealed class FileSizeTestDataLoader : BaseTestDataLoader
    {
        public override ICollection<BaseTestData> ExtractData(XElement xmlNode,
            ICollection<DataProperty> properties)
        {
            var result = new Collection<BaseData>();
            var allowedAttributeNames = new Collection<string> {"path", "size"};

            foreach (var node in xmlNode.Descendants()
                .Where(x => x.Name.LocalName.Equals(this.ElementName)))
            {
                var data = new FileSizeData
                {
                    Path = DataPropertyTool.ResolvePropertiesInString(allowedAttributeNames, properties,
                        XmlTools.GetNamedAttributeValue(node, "path", string.Empty)),
                    Size = int.Parse(XmlTools.GetNamedAttributeValue(node, "size", string.Empty))
                };

                AddCommonData(node, data, properties);
                result.Add(data);
            }

            return result;
        }
    }
}
```

*Figure 18: Test data loader example*
4.10.3 Test Data Class

The data class is an in memory representation of the test data. This class is used to send information from the loader class to the check class. The test data class needs to be marked with the same “DataType” attribute as the check class, as seen in Figure 19.

```csharp
namespace InstallerVerificationLibrary.Data
{
    using InstallerVerificationLibrary.Attribute;

    [DataType("FileSize")]
    public sealed class FileSizeData : BaseTestData
    {
        public string Path { get; set; }
        public int Size { get; set; }

        public override bool Equals(object obj)
        {
            var f1 = obj as FileSizeData;
            if (f1 == null) return false;
            return f1.Path.ToUpperInvariant() == Path.ToUpperInvariant() && f1.Size == Size;
        }

        public override int GetHashCode()
        {
            return base.GetHashCode();
        }

        public override string ToString()
        {
            return "File path:’" + Path + "’ File size:’" + Size + "’";
        }
    }
}
```

*Figure 19: Test data example*
4.11 Test scenario

To test that the implementation works correctly, three main installer scenarios need to be tested. All installer scenarios need to be executed on a machine with many processes running at the same time, this is to ensure that all environment noise is properly filtered out. Additionally, all installer test scenarios need to be executed with both an installer that behaves as expected and one that makes unexpected changes to both file system and registry.

Installer test scenarios

1. **Install**: ensure that an installation can be executed and tested. An test output can be seen in Figure 20
2. **Repair**: ensure that a repair can be executed and tested
3. **Uninstall**: ensure that an uninstallation can be executed and tested

![Figure 20: Install test execution output example](image-url)
5 Evaluation

The goal with this work was to find a simple, predictable and automated way to determine if a software installer has been installed correctly and no side effect is present on the system. The solution is based on a library that can be used in any testing framework with a human readable xml file format for the test data definition. A test case can be defined in six effective lines of code and executed as seen in Figure 21. Both the xml test data definition format and the few lines of code needed are aimed for easiness of usage. It is possible to extend the library for additional or more specific testing, simply by writing your own plugins. A plugin can be created with as few lines of code as seen in section 4.10.

All test scenarios from section 4.11 have been executed multiple times without failure.

```
[TestMethod]
public void Installer1_Install_ALLFeatures()
{
    var config = new SetupConfigInstaller1
    {
        TypeOfInstallation = TypeOfInstallation.Install,
        FeatureOne = true,
        FeatureTwo = true
    };
    var testBed = new MsiTestBed(config);
    Assert.IsTrue(testBed.Execute());
}
```

*Figure 21: Automated test cases example*
6 Discussion

The decision with the highest impact and also the biggest research effort was on evaluating different ways to collect file and registry changes. Out of eleven groups of solutions only three were fulfilling the requirements. In the end Process Monitor was picked due to the delimitation of .NET platform. It is a simple implementation but, it also added a big overhead both in execution time and memory usage. A kernel minifilter implementation would have been a more effective and faster way to do it.

Without a good way to reduce the amount of noise in the collected data, it is not possible to create a stable or predictable solution. The decision to use process id for filtering does reduce the noise significantly, but it also limits the solution to only cover changes done by the installation process or any of its child processes. Filtering based on process id makes the solution extremely predictable since the solution will never be affected by other process running in the system.

Simplicity of usage is a core element in the developed solution. This has been achieved by choosing to use xml based test data definition files that can be maintained by anyone. Using xml files for test data definition enables the recommended way to store both production code and its corresponding test code in the same source control system. This ensures that test and product code are always in sync, hence, it will be stable and predictable. Other solutions considered using external systems as a database, would not give this benefit.

Also, creating a generic library that can be used in any .Net testing framework, makes the solution quickly adaptable, since a developer can use their preferred and well known framework. Creating a test case using the solution can be done in a few lines of code, hence, it is fast to adopt and it lowers the possibility for errors. “The industry average number of bugs per thousand lines of code (kilo lines of code or KLOC) is typically estimated at about 15 to 50.” [55]. All of these elements make the solution simple to work with and all created test cases stable and predictable. A native DLL wrapped with a managed wrapper (to be used by .Net code) would enable using the solution in any programming or scripting language supporting API calls on a windows system.

Enabling the user of the library to add their own checks to the solution ensures that there is no limitation on how to test a software installer. This makes it simple for a software company to adapt the usage of the library in their own testing. It also makes it possible to integrate into almost any existing testing framework or tool.

This work is a big step forward compared to previous existing methods relying in manual testing or methods relying on different techniques where it is not possible to distinguish who did the change in the system. Even though there is still room for further work as mentioned in details in section 7.
7 Conclusions and future work

The outcome of this effort is a software installation verification framework that can be used in any testing framework. This framework can be used by anyone with basic programming skills and installation software domain knowledge. If this framework would be used by a company as part of their system development lifecycle, it would be unlikely that they would release a software installer that could not uninstall correctly or that would do unexpected changes to the end user machine. There is always a risk, the human factor, to take into account when it comes to defining the test data, since this solution can only verify that the state has changed to what is defined or not.

This effort is covering ways to collect file changes and registry changes to a good extent. This data have only been used in verifying items that are directly created by the installation process or any of its child processes. The natural next step would be to use the collected that to test indirect changes to the system based on the software installation. A good example: the installation of a service is not detected due to the service setup is happening in a process called by the installation software or one of its child processes. Another area of possible improvement is in the way the file and registry changes are collected. The actual implementation is not the most efficient from a process or memory point of view. An implementation of a kernel minifilter could help to improve this.
8 References


Appendix

A1: Using the solution

To use the solution in any existing or new installer test setup, the first thing needed is to decide what installer to be tested, and then place the file in a known location as in Figure 22. The recommended way is to add an additional build task, or use an existing one, to place the file as part of the build process.

![Image](image1.png)

Figure 22: Installation location

Next step is to create a test definition file based on the installer. It is recommended to store this file in the same source control as the actual installer code, to ensure that they are always in sync.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<msi id="Installer1">
  <include ref="CommonProperties.xml" />
  <features>
    <feature name="Feature1">
      <directory path="[INSTALLFOLDER]" />
      <file path="[INSTALLFOLDER]\Payload1.txt" />
      <file path="[INSTALLFOLDER]\EmptyFile.xml" />
      <registrykey key="HKCU\Software\Installer1" />
      <registryvalue key="HKCU\Software\Installer1" name="SomeIntegerValue" data="1" />
      <registryvalue key="HKCU\Software\Installer1" name="(Default)" data="Hi there" />
    </feature>
    <feature name="Feature2">
      <directory path="[INSTALLFOLDER]" />
      <file path="[INSTALLFOLDER]\PayLoadAssembly.exe" fileVersion="1.0.0.0" />
      <file path="[INSTALLFOLDER]\Client.exe"/>
      <startmenu allusers="true" name="New Client"/>
    </feature>
  </features>
  <properties>
    <property id="INSTALLFOLDER" type="Parameter" parameterID="INSTALLFOLDER" />
  </properties>
</msi>
```

![Image](image2.png)

Figure 23: Test data definition file
All installers need a setup configuration class that defines the behavior of the installer. The setup configuration needs to inherit from SetupConfigurationBaseMsi as seen in Figure 24, otherwise it will not be recognized by the solution. The setup configuration has two lists: the first one is named ParameterList containing all parameters used during installation and the second one named ComponentsList which contains all components. A parameter can be seen as a simple key value pair used to change the behavior of the installer. In Figure 23 we can see that a parameter is used to decide where to install all files. The same parameter is found in Figure 24 named InstallationFolderParameter. The component is used to select what features should be installed. Figure 23 has two features named “Feature1” and “Feature2” that are implemented in the setup configuration file as properties to simplify the usage.

```csharp
public class SetupConfigInstaller1 : SetupConfigBaseMsi
{
    private const string FeatureOneId = "Feature1";
    private const string FeatureTwoId = "feature2";
    private const string InstallFolderParameterId = "INSTALL_FOLDER";

    public SetupConfigInstaller1()
    {
        ParameterList.Add(new SetupConfigParameterData { Id = InstallFolderParameterId,
            Value = @"[ProgramFilesFolder]InstallationDirectory1" });
        ComponentList.Add(new SetupConfigComponentData { Id = FeatureOneId, Installed = true });
        ComponentList.Add(new SetupConfigComponentData { Id = FeatureTwoId, Installed = true });
        FilePathToMsiFile = @"Installers\Installer1.msi";
        FilePathToTestData = @"TestData\Installer1.xml";
        Msiid = "Installer1";
    }

    #region Features
    public bool FeatureOne
    {
        get { return GetComponent(FeatureOneId).Installed; }
        set { GetComponent(FeatureOneId).Installed = value; }
    }
    public bool FeatureTwo
    {
        get { return GetComponent(FeatureTwoId).Installed; }
        set { GetComponent(FeatureTwoId).Installed = value; }
    }
    #endregion

    #region Parameters
    public string InstallFolderParameter
    {
        get { return GetParameter(InstallFolderParameterId).Value; }
        set { GetParameter(InstallFolderParameterId).Value = value; }
    }
    #endregion
}
```

In Figure 25 the Microsoft unit test framework is used to implement a test case testing an installation scenario. The test case has three deployment items that copy the installer and the test data to the test execution location. They also copy procmon.exe to the test execution location, which is used to collect changes done to the system during test execution. The setup configuration in Figure 24 is used to define where the different
files can be found and what type of scenario is being tested, as seen in Figure 25. The actual execution and verification is done by the MsiTestBed class. The execute method returns true or false to indicate success or failure of the test.

```csharp
[TestMethod]
[DeploymentItem(@"C:\My Installers\Installer1.msi","Installers")]
[DeploymentItem(@"TestData", "TestData")]
[DeploymentItem(@"Procmon.exe")]
public void Installer1_Install_DefaultFeature()
{
    var config = new Installer1SetupConfig()
    {
        TypeOfInstallation = TypeOfInstallation.Install,
        FeatureOne = true,
        FeatureTwo = false
    };
    var testBed = new MsiTestBed(config);
    Assert.IsTrue(testBed.Execute());
}
```

Last thing to do is to actually run the test case and see the result. Figure 26 shows a test run of the test case from Figure 25.

![Test result output](image-url)