Security Policy

for FIPS 140-2 Validation

Microsoft Windows 8

Microsoft Windows Server 2012

Microsoft Windows RT

Microsoft Surface Windows RT

Microsoft Surface Windows 8 Pro

Microsoft Windows Phone 8

Microsoft Windows Storage Server 2012

Code Integrity (CI.DLL)

DOCUMENT INFORMATION

Version Number	1.2
Updated On	December 17, 2014

TABLE OF CONTENTS

1 Introduction

Code Integrity is a Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 feature that verifies the integrity of several key Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 binary image files as they are loaded into memory from the disk.

Code Integrity is not a general purpose cryptographic module. It is validated under FIPS 140-2 because it implements cryptographic algorithms and provides the integrity checks for the Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 general purpose cryptographic modules.

1.1 List of Cryptographic Module Binary Executables

CI.DLL – Version 6.2.9200 for Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8

1.2 Brief Module Description

Code Integrity is a dynamically-linked library used to verify the integrity of other binary executable code files.

1.3 Validated Platforms

The Code Integrity component listed in Section 1.1 was validated using the following machine configurations:

x86 Microsoft Windows 8 Enterprise – Dell Dimension C521 (AMD Athlon 64 X2 Dual Core)
x64 Microsoft Windows 8 Enterprise – Dell PowerEdge SC430 (Intel Pentium D without AES-NI)
x64-AES-NI Microsoft Windows 8 Enterprise – Intel Client Desktop (Intel Core i7 with AES-NI)
x64 Microsoft Windows Server 2012 – Dell PowerEdge SC430 (Intel Pentium D without AES-NI)
x64-AES-NI Microsoft Windows Server 2012 – Intel Client Desktop (Intel Core i7 with AES-NI)
ARMv7 Thumb-2 Microsoft Windows RT – NVIDIA Tegra 3 Tablet (NVIDIA Tegra 3 Quad-Core)
ARMv7 Thumb-2 Microsoft Windows RT – Qualcomm Tablet (Qualcomm Snapdragon S4)
ARMv7 Thumb-2 Microsoft Windows RT – Microsoft Surface Windows RT (NVIDIA Tegra 3 Quad-Core)
x64-AES-NI Microsoft Windows 8 Pro – Microsoft Surface Windows 8 Pro (Intel x64 Processor with AES-NI)
ARMv7 Thumb-2 Microsoft Windows Phone 8 – Windows Phone 8 (Qualcomm Snapdragon S4)
x64 Microsoft Windows Storage Server 2012 – Intel Maho Bay (Intel Core i7 without AES-NI)
x64-AES-NI Microsoft Windows Storage Server 2012 – Intel Maho Bay (Intel Core i7 with AES-NI)

Code Integrity maintains FIPS 140-2 validation compliance (according to FIPS 140-2 PUB Implementation Guidance G.5) on the following platforms:

x86 Microsoft Windows 8
x86 Microsoft Windows 8 Pro

x64 Microsoft Windows 8

x64 Microsoft Windows 8 Pro
x64 Microsoft Windows Server 2012 Datacenter

x64-AES-NI Microsoft Windows 8
x64-AES-NI Microsoft Windows 8 Pro
x64-AES-NI Microsoft Windows Server 2012 Datacenter

1.4 Cryptographic Boundary

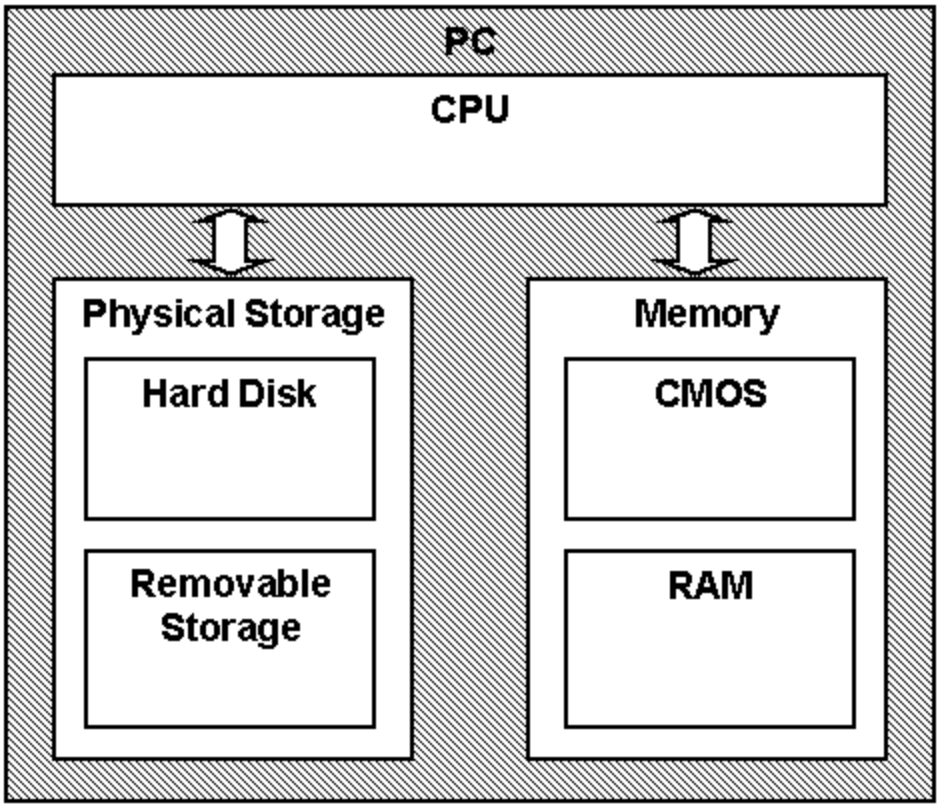
The cryptographic boundary for Code Integrity is defined as the enclosure of the computer system, on which Code Integrity is to be executed. The physical configuration of Code Integrity, as defined in FIPS-140-2, is multi-chip standalone.

2 Security Policy

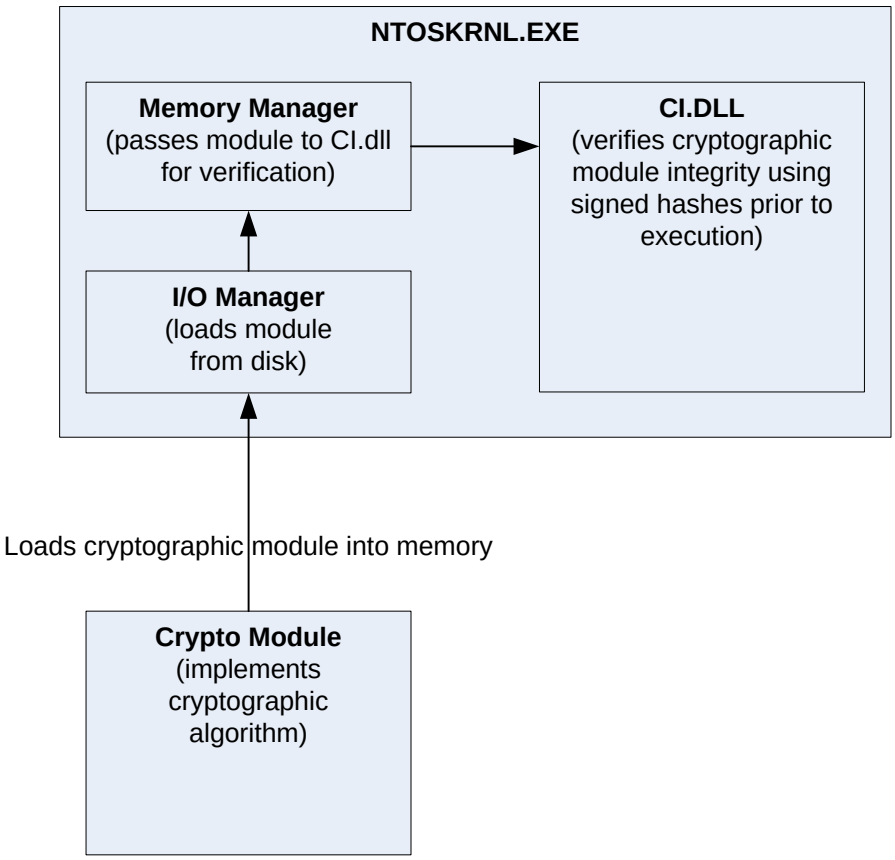
Code Integrity is considered to be in a FIPS mode of operation when the following rules are followed:

- Code Integrity is supported on Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8.
- Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 are operating systems supporting a “single user” mode where there is only one interactive user during a logon session.
- Code Integrity is only in the “Approved mode of operation” when Windows is booted normally, meaning Debug mode has not been enabled and Driver Signing enforcement has not been disabled.
- The Debug mode status and Driver Signing enforcement status can be viewed by using the bcdedit tool.
- Code Integrity operates in FIPS mode of operation only when used with the FIPS approved version of Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 Winload OS Loader (winload.exe) validated to FIPS 140-2 under Cert. #1896 operating in FIPS mode.

The following diagram illustrates the master components of the Code Integrity module:



The following diagram illustrates the interaction of Code Integrity with other cryptographic modules:



- Code Integrity's main service is to verify the integrity of digitally signed drivers and components within the computer (such as bcryptprimitives.dll, rsaenh.dll, dssenh.dll). In addition to this service, Code Integrity also provides status services. These status services indicate whether the aforementioned integrity checks passed.
- All services implemented within Code Integrity are available to the User and Crypto officer roles. The User and Crypto officer roles are assumed by the operating system or application processes that will invoke binary image verification in CI.dll. The Window Memory Manager is an example.
- Code Integrity verifies the integrity of the Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 general purpose cryptographic modules using the following FIPS-140-2 Approved algorithms.
    - RSA PKCS#1 (v1.5) verify with public key
    - SHA-1 hash
    - SHA-256 hash
    - SHA-384 hash
    - SHA-512 hash

2.1 FIPS 140-2 Approved Algorithms

Code Integrity implements the following FIPS-140-2 Approved algorithms:

- RSA PKCS#1 (v1.5) digital signature verification (Cert. # 1132)
- SHA-1 hash (Cert. # 1903)
- SHA-256 hash (Cert. # 1903)
- SHA-384 hash (Cert. # 1903)
- SHA-512 hash (Cert. # 1903)

2.2 Non-Approved Algorithms

Code Integrity also includes a legacy implementation of MD5. MD5 is only used for backwards compatibility to verify the RSA signature over the file digest and certificate chains. MD5 is not allowed for use in file digests, which require a SHA-1 hash as the minimum.

2.3 Cryptographic Bypass

Cryptographic bypass is not supported by Code Integrity.

2.4 Machine Configurations

Code Integrity was tested using the machine configurations listed in Section 1.3 - Validated Platforms.

3 Integrity Chain of Trust

3.1 Conventional BIOS and UEFI without Secure Boot Enabled

Boot Manager is the start of the chain of trust. It cryptographically checks its own integrity during its startup. It then cryptographically checks the integrity of the Windows OS Loader (Winload.exe) before starting it. The Windows OS Loader checks the integrity of Code Integrity, which is protected by an RSA signature with a 2048-bit key and SHA-256 message digest, before loading it into memory. Code Integrity is used to verify the origin and integrity of Windows system binaries before they are loaded into memory and executed. Code Integrity also ensures kernel mode drivers are appropriately signed. When User Mode Code Integrity (UMCI) is enabled, Code Integrity ensured that all binaries are appropriately signed.

3.2 UEFI with Secure Boot Enabled

On UEFI systems with Secure Boot enabled, Boot Manager is still the OS binary from which the integrity of all other OS binaries is rooted, and it does cryptographically check its own integrity. However, Boot Manager's integrity is also checked and verified by the UEFI firmware, which is the root of trust on Secure Boot enabled systems.

4 Ports and Interfaces

4.1 Code Integrity export functions

The following list contains all the functions exported by Code Integrity to its callers. Note that Code Integrity is not callable outside the kernel. They are explained further in the subsequent subsections.

- CiInitialize()
  - CiValidateImageHeader()
  - CiValidateImageData()
  - CiQueryInformation()
  - CiQueryImageSignature()
  - CiImportRoots()
  - CiGetFileCache()
  - CiSetFileCache()
  - CiHashMemorySha256()
- CiGetPEInformation()
- CiVerifyHashInCatalog()
- CiCheckSignedFile()
- CiFindPageHashesInCatalog()
- CiFindPageHashesInSignedFile()
- CiFreePolicyInfo()

4.1.1 CiInitialize()

CiInitialize() is the function exported by Code Integrity for initializing the image file integrity validation capability of Code Integrity.

As the power-on (startup) function of Code Integrity, CiInitialize() conducts the following power-on (startup) self-tests.

- SHS (SHA-1) Known Answer Test
- SHS (SHA-256) Known Answer Test
- SHS (SHA-512) Known Answer Test
- RSA verify using a verify test with a Known Signatures of the PKCS#1 v1.5 format:
  - RSA signature with 1024-bit key and SHA-1 message digest
  - RSA signature with 2048-bit key and SHA-256 message digest

If a self-test fails, CiInitialize() returns STATUS_INVALID_IMAGE_HASH. On the other hand, after the successful initialization, CiInitialize() returns a callback structure consisting of the following functions. A caller subsequently can use these functions to obtain the image file integrity validation service from Code Integrity.

- CiValidateImageHeader()
- CiValidateImageData()
- CiQueryInformation()
- CiQueryImageSignature()
- CiImportRoots()

- CiGetFileCache()
- CiSetFileCache()
- CiHashMemorySha256()

4.1.1.1 CiValidateImageHeader()

When a caller (such as the Memory Manager) wants to obtain the set of trusted per-page hashes of an image file, it calls CiValidateImageHeader(). Trusted per-page hashes can use the following algorithms:

- SHS (SHA-1)
- SHS (SHA-256)
- SHS (SHA-384)
- SHS (SHA-512)

In the case of a Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 general purpose cryptographic module (namely, bcryptprimitives.dll, rsaenh.dll, or dssenh.dll), if CiValidateImageHeader() does not find the set of trusted per-page hashes for the cryptographic module, then CiValidateImageHeader() verifies the full cryptographic module image by verifying a trusted file hash. The trusted file hash may be:

- SHS (SHA-1)
- SHS (SHA-256)
- SHS (SHA-384)
- SHS (SHA-512)

If this validation process fails, the cryptographic module is not valid. Subsequently, the Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 Memory Manager does not load any page of the cryptographic module.

Both the trusted file image hash and trusted page hashes are signed using the RSA signature algorithm with PKCS#1 v1.5 padding.

Code Integrity has a different verification procedure for kernel mode crypto modules that are loaded into memory all at once (not in a per-page fashion as the other user mode general purpose crypto modules). As a result, when CiValidateImageHeader() is called by the memory manager, the CI_VALIDATE_DRIVER_IMAGE flag is set, and the entire image is validated by verifying a trusted image hash. This is similar to the user mode module verification when page hashes are not present.

4.1.1.2 CiValidateImageData()

After calling CiValidateImageHeader() to obtain the set of trusted per-page hashes of an image file, a caller (such as the Memory Manager) would want to know the integrity of a page of the image file. The caller uses CiValidateImageData() to check the page integrity.

If the computed hash matches the identified trusted hash, then CiValidateImageData confirms the integrity of the page. Otherwise, CiValidateImageData() returns STATUS_INVALID_IMAGE_HASH. The

Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 Memory Manager does not load invalid pages.

4.1.1.3 CiQueryInformation()

Returns state data about the enforcement of Code Integrity. Whether CI is being enforced and whether test signing is enabled.

4.1.1.4 CiQueryImageSignature()

Returns whether a previously validated file is Windows signed (signing certificate chains to Microsoft Root and the Windows EKU). This check was done during a previous validation, and this function is just returning a cached result.

4.1.1.5 CiImportRoots()

Imports public keys that are used as trusted CAs for validation of user mode components.

4.1.1.6 CiGetFileCache()

For an input file, returns the previously validated signature level (MSFT, Windows, Authenticode) and the thumbprint of the signing certificate. This check was done during a previous validation, and this function is just returning a cached result.

4.1.1.7 CiSetFileCache()

For a verified file, saves the signature level and thumbprint of the signing certificate. If the file was not previouisly verified, it will verify the file against either its embedded signature or a system catalog.

4.1.1.8 CiHashMemorySha256()

Passes supplied data to CI's SHA256 implementation and returns the SHA256 hash of that data.

4.1.2 CiGetPEInformation

Creates an encrypted channel between the caller and CI. It is used as part of protected media path DRM, and allows information about kernel drivers and user mode binaries loaded into protected processes to be returned to the caller.

4.1.3 CiVerifyHashInCatalog

For an input Authenticode file digest, validates that the digest is contained within a verified system catalog. It optionally returns information about the catalog.

4.1.4 CiCheckSignedFile

For an input Authenticode file digest and an Authenticode signature, verifies that the digest is in the signature and that the signature validates. It optionally returns information about the signature.

4.1.5 CiFindPageHashesInCatalog

For an input Authenticode digest of the first page of a PE image, validates that the digest is contained within a verified system catalog. It optionally returns information about the catalog.

4.1.6 CiFindPageHashesInSignedFile

For an input Authenticode digest of the first page of a PE image and an Authenticode signature, verifies that the digest is in the signature and that the signature validates. It optionally returns information about the signature.

4.1.7 CiFreePolicyInfo

Frees memory allocated by the CiVerifyHashInCatalog(), CiCheckSignedFile(), CiFindPageHashesInCatalog(), and CiFindPageHashesInSignedFile() functions.

4.2 Control Input Interface

The Control Input Interface for Code Integrity consists of the three CI export functions. Options for control operations are passed as input parameters to the CI export functions. The SecureRequired parameter in CiValidateImageHeader() is the only control option provided by Code Integrity in the Control Input Interface.

4.3 Status Output Interface

The Status Output Interface for Code Integrity also consists of the three CI export functions. For each function, the status information is returned to the caller as the return value (e.g. STATUS_SUCCESS, STATUS_UNSUCCESSFUL, STATUS_INVALID_IMAGE_HASH) from the function.

4.4 Data Input Interface

The Data Input Interface for Code Integrity also consists of the three CI export functions. Data and options are passed to the interface as input parameters to the CI export functions. Data Input is kept separate from Control Input by passing Data Input in separate parameters from Control Input.

4.5 Data Output Interface

The Data Output Interface for Code Integrity also consists of the three CI export functions. For CiInitialize(), data is returned to its caller as the Callbacks output parameter. For CiValidateImageHeader(), data is returned to its caller as the SePool output parameter.

5 Specification of Roles

Code Integrity supports both User and Cryptographic Officer roles (as defined in FIPS-140-2). Both roles have access to all services implemented in Code Integrity through a caller component running in the kernel mode. The module does not provide authentication, as such both roles are implicitly assumed when the services exported by the module are invoked.

5.1 Maintenance Roles

Maintenance roles are not supported.

5.2 Multiple Concurrent Interactive Operators

There is only one interactive operator during a logon session. Multiple concurrent interactive operators sharing a logon session are not supported.

5.3 Show Status Services

The User and Cryptographic Officer roles have the same Show Status functionality, which is, for each function, the status information is returned to the caller as the return value from the function.

5.4 Self-Test Services

The User and Cryptographic Officer roles have the same Self-Test functionality, which is described in Section 10 Self-Tests.

5.5 Service Inputs / Outputs

The User and Cryptographic Officer roles have service inputs and outputs as specified in Section 0 Ports and Interfaces.

6 Services

Code Integrity does not offer any other services, operations, or functions that can be externally invoked.

7 Operational Environment

The operational environment for Code Integrity is Windows 8, Windows RT, Windows Server 2012, Windows Storage Server 2012, and Windows Phone 8 running on the hardware listed in Section 1.3 - Validated Platforms.

8 Authentication

Code Integrity does not implement any authentication services. The User and Cryptographic Officer roles are assumed implicitly by booting the Windows operating system. There are Code Integrity libraries that run before boot in Winload.exe and Winresume.exe.The CI.DLL is loaded in the kernel as part of the memory management path.

9 Cryptographic Key Management

Code Integrity does not handle security-relevant information such as secret and private cryptographic key, authentication data, nor any other protected information. Hence, there is no operation related to any of the below.

- Key generation
- Key output
- Key storage

The only cryptographic keys the module supports are the RSA PKCS#1 public keys used to verify integrity. These public keys are accessible by both approved roles. Due to such simplicity, an access control policy table is not included in this document. The public keys are stored on the hard-drive. Zeroization is performed by deleting the Code Integrity module, ntph.cat, and ntpe.cat files.

9.1 Cryptographic Keys

The Code Integrity crypto module uses the following cryptographic keys:

Cryptographic Key	Key Description
Asymmetric Public keys	Keys used for RSA PKCS#1 (v1.5) verification

9.2 Critical Security Parameters

The Code Integrity crypto module does not contain any Critical Security Parameters (CSPs).

9.3 Security Relevant Data Items

The Code Integrity crypto module does not contain Security Relevant Data Items (SRDIs).

9.4 Access Control Policy

The Code Integrity crypto module does not contain CSPs nor SRDIs that would require access controls.

10 Self-Tests

10.1 Power-On Self-Tests

Code Integrity performs the following power-on (startup) self-tests when a caller calls its CiInitialize():

- SHS (SHA-1) Known Answer Test
- SHS (SHA-256) Known Answer Test
- SHS (SHA-512) Known Answer Test
- RSA verify using a verify test with a Known Signature of the PKCS#1 v1.5 format with both 1024-bit keys with SHA1 digest and 2048-bit keys with SHA-256 digest.

The integrity of Code Integrity itself is protected by an RSA signature with a 2048-bit key and SHA-256 message digest, which is verified by Winload.exe before Code Integrity is loaded into memory. If the self-test fails, the module will not load and status will be returned. If the status is not STATUS_SUCCESS, then that is the indicator a self-test failed.

10.2 Conditional Self-Tests

Code Integrity does not perform conditional self-tests.

11 Design Assurance

The secure installation, generation, and startup procedures of this cryptographic module are part of the overall Windows 8, Windows RT, Windows Server 2012, and Windows Storage Server 2012 operating system secure installation, configuration, and startup procedures. After the operating system has been installed, it must be configured by enabling the "System cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing" policy setting followed by restarting the system. This procedure is all the crypto officer and user behavior necessary for the secure operation of this cryptographic module.

Windows Phone 8 does not use the same installation, configuration, and startup procedures as the Windows operating system on a computer, but rather, is securely installed and configured by the cellular telephone carrier.

The procedures required for maintaining security while distributing and delivering versions of a cryptographic module to authorized operators are:

1. The secure distribution method is via the physical medium for product installation delivered by Microsoft Corporation, which is a DVD in the case of Windows 8 and Windows Server 2012. In the case of Windows RT, Surface Windows RT, Surface Windows 8 Pro, Windows Phone 8, and Windows Storage Server 2012, the cryptographic module is already installed at the factory and is only distributed with the hardware.
2. An inspection of authenticity of the physical medium can be made by following the guidance at this Microsoft web site: http://www.microsoft.com/en-us/howtotell/default.aspx
3. The installed version of Windows 8, Windows RT, Windows Server 2012, and Windows Storage Server 2012 must be verified to match the version that was validated. See Appendix A for details on how to do this.

For Windows Updates, the client only accepts binaries signed by Microsoft certificates. The Windows Update client only accepts content whose SHA-2 hash matches the SHA-2 hash specified in the metadata. All metadata communication is done over a Secure Sockets Layer (SSL) port. Using SSL ensures that the client is communicating with the real server and so prevents a spoof server from sending the client harmful requests. The version and digital signature of new cryptographic module releases must be verified to match the version that was validated. See Appendix A for details on how to do this.

12 Mitigation of Other Attacks

The following table lists the mitigations of other attacks for this cryptographic module:

Algorithm	Protected Against	Mitigation	Comments
SHA1	Timing Analysis Attack	Constant Time Implementation	
	Cache Attack	Memory Access pattern is independent of any confidential data	
SHA2	Timing Analysis Attack	Constant Time Implementation	
	Cache Attack	Memory Access pattern is independent of any confidential data	

13 Additional Details

For the latest information on Microsoft Windows, check out the Microsoft web site at:

http://windows.microsoft.com

For more information about FIPS 140 evaluations of Microsoft products, please see:

http://technet.microsoft.com/en-us/library/cc750357.aspx

14 Appendix A – How to Verify Windows Versions and Digital Signatures

14.1 How to Verify Windows Versions

The installed version of Windows 8, Windows RT, Windows Server 2012, and Windows Storage Server 2012 must be verified to match the version that was validated using one of the following methods:

1. The ver command
   a. From Start, open the Search charm.
   b. In the search field type "cmd" and press the Enter key.
   c. The command window will open with a "C:\>" prompt.
   d. At the prompt, type "ver" and press the Enter key.
   e. You should see the answer "`Microsoft Windows [Version 6.2.9200]`".
2. The systeminfo command
   a. From Start, open the Search charm.
   b. In the search field type "cmd" and press the Enter key.
   c. The command window will open with a "C:\>" prompt.
   d. At the prompt, type "systeminfo" and press the Enter key.
   e. Wait for the information to be loaded by the tool.
   f. Near the top of the output, you should see:

```
OS Name:                   Microsoft Windows 8 Enterprise
OS Version:                6.2.9200 N/A Build 9200
OS Manufacturer:           Microsoft Corporation
```

If the version number reported by the utility matches the expected output, then the installed version has been validated to be correct.

14.2 How to Verify Windows Digital Signatures

After performing a Windows Update that includes changes to a cryptographic module, the digital signature and file version of the binary executable file must be verified. This is done like so:

1. Open a new window in Windows Explorer.
2. Type “C:\Windows\” in the file path field at the top of the window.
3. Type the cryptographic module binary executable file name (for example, “CNG.SYS”) in the search field at the top right of the window, then press the Enter key.
4. The file will appear in the window.
5. Right click on the file’s icon.
6. Select Properties from the menu and the Properties window opens.
7. Select the Details tab.
8. Note the File version Property and its value, which has a number in this format: x.x.xxxx.xxxxx.
9. If the file version number matches one of the version numbers that appear at the start of this security policy document, then the version number has been verified.
10. Select the Digital Signatures tab.
11. In the Signature list, select the Microsoft Windows signer.
12. Click the Details button.
13. Under the Digital Signature Information, you should see: “This digital signature is OK.” If that condition is true then the digital signature has been verified.