



Microsoft Windows

FIPS 140 Validation

Microsoft Windows Server 2019

Microsoft Azure Stack Edge

Microsoft Azure Stack Hub

Microsoft Azure Stack Edge Rugged

Non-Proprietary

Security Policy Document

Version Number	1.2
Updated On	September 8, 2023

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. This work is licensed under the Creative Commons Attribution-NoDerivs-NonCommercial License (which allows redistribution of the work). To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd-nc/1.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2023 Microsoft Corporation. All rights reserved.

Microsoft, Windows, the Windows logo, Windows Server, and BitLocker are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Version History

Version	Date	Summary of Changes
1.0	November 4, 2020	Draft sent to NIST CMVP
1.1	November 3, 2022	Updates in response to NIST feedback
1.2	September 8, 2023	Updates in response to NIST feedback, updated bounded module certificates

TABLE OF CONTENTS

<u>SECURITY POLICY DOCUMENT</u>	<u>1</u>
<u>1 INTRODUCTION</u>	<u>7</u>
1.1 LIST OF CRYPTOGRAPHIC MODULE BINARY EXECUTABLES	7
1.2 VALIDATED PLATFORMS	7
<u>2 CRYPTOGRAPHIC MODULE SPECIFICATION</u>	<u>9</u>
2.1 CRYPTOGRAPHIC BOUNDARY	9
2.2 FIPS 140-2 APPROVED ALGORITHMS	9
2.3 NON-APPROVED ALGORITHMS	9
2.4 FIPS 140-2 APPROVED ALGORITHMS FROM BOUNDED MODULES	10
2.5 CRYPTOGRAPHIC BYPASS	10
2.6 HARDWARE COMPONENTS OF THE CRYPTOGRAPHIC MODULE	10
<u>3 CRYPTOGRAPHIC MODULE PORTS AND INTERFACES</u>	<u>11</u>
3.1 CODE INTEGRITY EXPORT FUNCTIONS	11
3.1.1 CiInitialize	11
3.1.2 CiGetPEInformation	12
3.1.3 CiVerifyHashInCatalog	12
3.1.4 CiCheckSignedFile	12
3.1.5 CiFindPageHashesInCatalog	12
3.1.6 CiFindPageHashesInSignedFile	12
3.1.7 CiFreePolicyInfo	13
3.1.8 CiSetTrustedOriginClaimId	13
3.1.9 CiValidateFileObject	13
3.2 CODE INTEGRITY CALLBACK FUNCTIONS	13
3.2.1 CiValidateImageHeader	13
3.2.2 CiValidateImageData	14
3.2.3 CiQueryInformation	14
3.2.4 CiSetFileCache	14
3.2.5 CiGetFileCache	14
3.2.6 CiHashMemory()	14
3.2.7 KappIsPackageFile	15
3.2.8 CiCompareSigningLevels	15
3.2.9 CiValidateFileAsImageType	15

3.2.10	CIREGISTERSIGNINGINFORMATION	15
3.2.11	CIUNREGISTERSIGNINGINFORMATION	15
3.2.12	CIINITIALIZEPOLICY	15
3.2.13	CIPQUERYPOLICYINFORMATION	15
3.2.14	CIVALIDATEDYNAMICCODEPAGES.....	15
3.2.15	SIPOLICYQUERYSECURITYPOLICY	15
3.2.16	CIREVALIDATEIMAGE	15
3.2.17	CISSETUNLOCKINFORMATION	15
3.2.18	CIGETBUILDEXPIRYTIME.....	15
3.2.19	CIGETSTRONGIMAGEREFERENCE	15
3.2.20	CIRELEASECONTEXT	16
3.2.21	CIHVCISETIMAGEBASEADDRESS	16
3.3	CONTROL INPUT INTERFACE	16
3.4	STATUS OUTPUT INTERFACE.....	16
3.5	DATA INPUT INTERFACE	16
3.6	DATA OUTPUT INTERFACE	16
4	<u>ROLES, SERVICES AND AUTHENTICATION</u>	<u>16</u>
4.1	ROLES.....	16
4.2	SERVICES	16
4.3	AUTHENTICATION	18
5	<u>FINITE STATE MODEL.....</u>	<u>18</u>
5.1	SPECIFICATION.....	18
6	<u>OPERATIONAL ENVIRONMENT.....</u>	<u>19</u>
6.1	SINGLE OPERATOR.....	19
6.2	CRYPTOGRAPHIC ISOLATION.....	19
6.3	INTEGRITY CHAIN OF TRUST	20
7	<u>CRYPTOGRAPHIC KEY MANAGEMENT</u>	<u>22</u>
8	<u>SELF-TESTS</u>	<u>22</u>
9	<u>DESIGN ASSURANCE.....</u>	<u>22</u>

10 **MITIGATION OF OTHER ATTACKS.....23**

11 **SECURITY LEVELS23**

12 **ADDITIONAL DETAILS24**

13 **APPENDIX A – HOW TO VERIFY WINDOWS VERSIONS AND DIGITAL SIGNATURES25**

13.1 **HOW TO VERIFY WINDOWS VERSIONS25**

13.2 **HOW TO VERIFY WINDOWS DIGITAL SIGNATURES25**

1 Introduction

Code Integrity (CI) verifies the integrity of Windows executable files as they are loaded into memory from storage. Code Integrity is implemented in a Dynamic Link Library (DLL) file, CI.DLL.

The Secure Kernel Code Integrity cryptographic module is closely related to Code Integrity, and, depending on the hardware and Windows configuration, will also validate system and application binaries.

Two Windows configuration options dictate whether Code Integrity or Secure Kernel Code Integrity are used to verify a binary image:

- Virtual Secure Mode (VSM), also known as Core Isolation: Windows can use the Hypervisor to start an execution environment, called the Secure Kernel, that can enforce additional security rules. When VSM is configured, Secure Kernel Code Integrity verifies the integrity of critical user-mode modules such as BCRYPTPRIMITIVES.DLL instead of the Code Integrity module.
- Hypervisor Code Integrity (HVCI), also known as Memory Integrity: This feature depends on VSM. When enabled, all drivers loaded into the Windows kernel are integrity verified by Secure Kernel Code Integrity.

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 general-purpose cryptographic modules. For the purpose of this validation, Code Integrity is classified as a Software cryptographic module.

This Security Policy Document assumes that the following prerequisites are available:

- UEFI Secure Boot is available and enabled. If UEFI Secure Boot is disabled, a local user can perform kernel debugging, which is not an Approved mode of operation.

1.1 List of Cryptographic Module Binary Executables

Code Integrity cryptographic module contains the following binaries:

- CI.DLL

The Windows builds covered by this validation are:

- Windows Server 2019 build 10.0.17763.10021 and 10.0.17763.10127

1.2 Validated Platforms





The editions covered by this validation are:

- Windows Server 2019 Datacenter Core

Code Integrity was validated using the combination of computers and Windows operating system editions specified in the table below.

All the computers for Windows Server and Windows Server listed in the table below are all 64-bit Intel architecture.

Table 1 Validated Platforms

Computer	Windows Server 2019 Datacenter Core	Processor Image
Microsoft Azure Stack Edge - Dell XR2 - Intel Xeon Silver 4114	√	 <p>wikichip.org</p>
Microsoft Azure Stack Hub - Dell PowerEdge R640 - Intel Xeon Gold 6230	√	 <p>wikichip.org</p>
Microsoft Azure Stack Hub - Dell PowerEdge R840 - Intel Xeon Platinum 8260	√	 <p>wikichip.org</p>
Microsoft Azure Stack Edge Rugged - Rugged Mobile Appliance – Intel Xeon D-1559	√	 <p>wikichip.org</p>

2 Cryptographic Module Specification

Code Integrity is a multi-chip standalone module that operates in FIPS-approved mode during normal operation of the computer and Windows operating system.

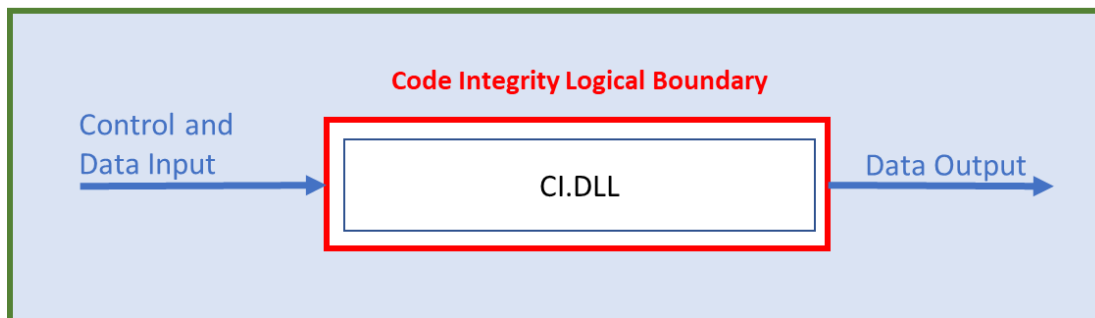
The following configurations and modes of operation will cause Code Integrity to operate in a non-approved mode of operation:

- Boot Windows in Debug mode
- Boot Windows with Driver Signing disabled

2.1 Cryptographic Boundary

The software binary that comprises the cryptographic boundary for Code Integrity is CI.DLL.

Physical Boundary – General Purpose Computer



2.2 FIPS 140-2 Approved Algorithms

Code Integrity implements the following FIPS 140-2 Approved algorithms: ¹

Table 2 Approved Algorithms

Algorithm	Windows Server 2019 build 10.0.17763.10021	Windows Server 2019 build 10.0.17763.10127
FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 1024, 2048, and 3072 moduli; supporting SHA-1, SHA-256, SHA-384, and SHA-512	#C1577	#C2044
FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512	#C1577	#C2044

2.3 Non-Approved Algorithms

Code Integrity only implements approved algorithms.

¹ This module may not use some of the capabilities described in each CAVP certificate.

2.4 FIPS 140-2 Approved Algorithms from Bounded Modules

A bounded module is a FIPS 140 module which provides cryptographic functionality that is relied on by a downstream module. As described in the [Integrity Chain of Trust](#) section, Code Integrity depends on the following modules and algorithms:

The Windows OS Loader for Windows Server version 1809 build 10.0.17763.10021 (module certificate [#4545](#)) provides

- CAVP certificates #C1586 (Windows Server 2019) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #C1577 (Windows Server 2019) for FIPS 180-4 SHS SHA-256

The Windows OS Loader for Windows Server version 1809 build 10.0.17763.10127 (module certificate [#4545](#)) provides

- CAVP certificates #C2052 (Windows Server 2019) for FIPS 186-4 RSA PKCS#1 (v1.5) digital signature verification with 2048 moduli; supporting SHA-256
- CAVP certificates #C2044 (Windows Server 2019) for FIPS 180-4 SHS SHA-256

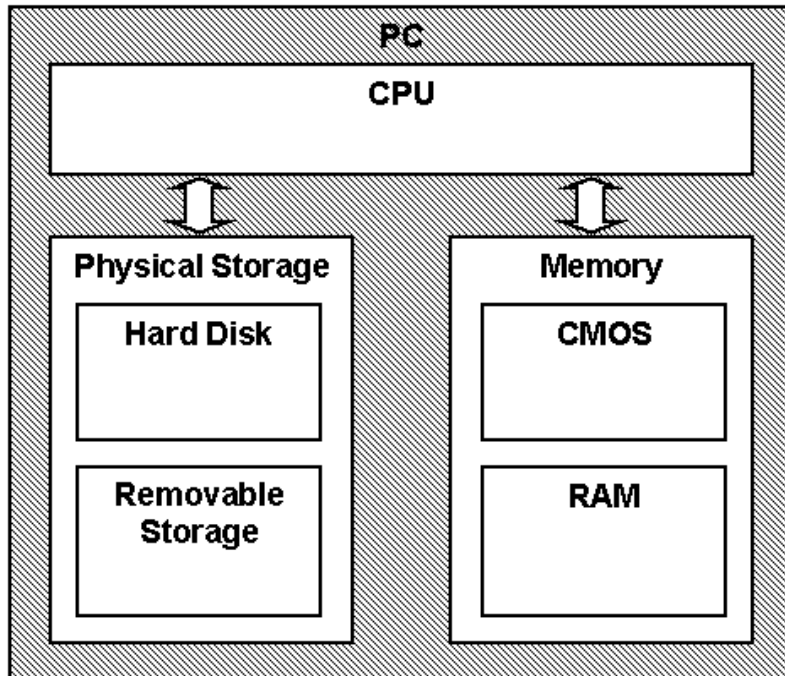
Note that the validated platforms listed in section 1.2 include processors that support AES-NI. This module does not implement AES, but the bounded modules may implement AES and, therefore, use AES-NI.

2.5 Cryptographic Bypass

Cryptographic bypass is not supported by Code Integrity.

2.6 Hardware Components of the Cryptographic Module

The physical boundary of the module is the physical boundary of the computer that contains the module. The following diagram illustrates the hardware components used by the Code Integrity module:



3 Cryptographic Module Ports and Interfaces

3.1 Code Integrity Export Functions

All the functions exported by Code Integrity to kernel-mode callers are listed below. Code Integrity is not callable outside the kernel. The exported functions are explained further in the subsequent subsections.

- CiInitialize
- CiGetPEInformation
- CiVerifyHashInCatalog
- CiCheckSignedFile
- CiFindPageHashesInCatalog
- CiFindPageHashesInSignedFile
- CiFreePolicyInfo
- CiSetTrustedOriginClaimId
- CiValidateFileObject

3.1.1 CiInitialize

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

See [Self-Tests](#) for information regarding cryptographic self-tests.

If the self-tests succeed, CiInitialize() returns a callback structure consisting of the following binary executable file integrity validation functions.

- CiValidateImageHeader
- CiValidateImageData
- CiQueryInformation
- CiSetFileCache
- CiGetFileCache
- CiHashMemory
- KappxIsPackageFile
- CiCompareSigningLevels
- CiValidateFileAsImageType
- CiRegisterSigningInformation
- CiUnregisterSigningInformation
- CiInitializePolicy
- CiQueryPolicyInformation
- CiValidateDynamicCodePages
- SiPolicyQuerySecurityPolicy
- CiRevalidateImage
- CiSetUnlockInformation
- CiGetBuildExpiryTime

And if HVCI is enabled, also:

- CiGetStrongImageReference
- CiReleaseContext
- CiHvciSetImageBaseAddress

3.1.2 CiGetPEInformation

This function returns system configuration data which is related to the protected media subsystem.

3.1.3 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.

3.1.4 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.

3.1.5 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.

3.1.6 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.

3.1.7 CiFreePolicyInfo

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

3.1.8 CiSetTrustedOriginClaimId

This function is invoked by Appid.sys when an AppLocker policy is being processed.

3.1.9 CiValidateFileObject

Verifies the signature of a file object and returns the policy info along with the timestamp and signing time.

3.2 Code Integrity Callback Functions

The following functions are not exported, but are accessed via a callback structure provided by the CiInitialize function. These functions are also explained in subsequent subsections.

- CiValidatImageHeader
- CiValidatImageData
- CiQueryInformation
- CiSetFileCache
- CiGetFileCache
- CiHashMemory
- KappxIsPackageFile
- CiCompareSigningLevels
- CiValidateFileAsImageType
- CiRegisterSigningInformation
- CiUnregisterSigningInformation
- CiInitializePolicy
- CipQueryPolicyInformation
- CiValidateDynamicCodePages
- SIPolicyQuerySecurityPolicy
- CiRevalidatImage
- CiSetUnlockInformation
- CiGetBuildExpiryTime

And if HVCI is enabled, also:

- CiGetStrongImageReference
- CiReleaseContext
- CiHvciSetImageBaseAddress

3.2.1 CiValidatImageHeader

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

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

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 module is not valid and the module is not loaded.

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

3.2.2 `CiValidateImageData`

After calling `CiValidateImageHeader` to obtain the set of trusted per-page hashes of an image file, `CiValidateImageData()` is used to check the integrity of each page by computing the hash value of the page.

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

3.2.3 `CiQueryInformation`

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

3.2.4 `CiSetFileCache`

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

3.2.5 `CiGetFileCache`

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

3.2.6 `CiHashMemory()`

Passes supplied data to `MinCrypK_HashMemory` and returns the hash of that data.

3.2.7 KappxIsPackageFile

This routine takes an input file object and parses out the full package name associated with the corresponding package. Package association is established based on the normalized path corresponding to the file object.

3.2.8 CiCompareSigningLevels

This routine determines if the source signing level is applicable for the target. For example, a source of Microsoft is not valid for a target of Windows, but vice-versa is valid.

3.2.9 CiValidateFileAsImageType

This routine determines if a PE file is signed appropriately for the specified image type. The file must be mapped as a view to a data section or be a copy of that view.

3.2.10 CiRegisterSigningInformation

This routine sets a signer, in addition to those already configured, for a specified signing level. Only those levels that are enabled for runtime configuration will accept signers supplied to this routine.

3.2.11 CiUnregisterSigningInformation

This routine removes a previous registered runtime signing information. Once a registration handle has been unregistered, it must be discarded.

3.2.12 CiInitializePolicy

This routine is called to get the configuration of CI for this boot and return the list of address ranges to be protected by Patch Guard.

3.2.13 CipQueryPolicyInformation

This routine returns information about Code Integrity policy state.

3.2.14 CiValidateDynamicCodePages

This routine validates the contents of code pages generated dynamically.

3.2.15 SiPolicyQuerySecurityPolicy

This routine queries the secure setting for specific provider's <Key,Value> pair.

3.2.16 CiRevalidateImage

This routine determines if previously validated images must be validated again.

3.2.17 CiSetUnlockInformation

This function sets unlock information for Code Integrity.

3.2.18 CiGetBuildExpiryTime

This routine determines the expiry time of the build. Zero time means the build never expires, which is true for production and test builds. Flight builds expire when the certificate that signs CI expires.

3.2.19 CiGetStrongImageReference

This routine returns the handle to a secure image.

3.2.20 CiReleaseContext

This routine closes a validation context.

3.2.21 CiHvciSetImageBaseAddress

This routine changes the base address of an image that is using secure relocations.

3.3 Control Input Interface

The SecureRequired parameter in CiValidateImageHeader() is the only control option provided by Code Integrity in the Control Input Interface.

3.4 Status Output Interface

The Status Output Interface for Code Integrity consists of the exported and callable functions listed in [Code Integrity 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.

3.5 Data Input Interface

The Data Input Interface for Code Integrity is the exported and callable functions listed in [Code Integrity export functions](#) with the exception of the initialization and status functions. Data and options are passed to the interface as input parameters to the CI export functions.

3.6 Data Output Interface

The Data Output Interface for Code Integrity also consists of most of the exported and callable functions listed in [Code Integrity export functions](#) with the exception of the initialization and status functions. Data is returned to the function's caller via output parameters.

4 Roles, Services and Authentication

4.1 Roles

Code Integrity is a library used solely by the Windows kernel and does not interact with the user through any service. The module's functions are fully automatic and not configurable. FIPS 140 validations define formal "User" and "Cryptographic Officer" roles. Both roles can use any Code Integrity service.

4.2 Services

Code Integrity's services are:

1. **Verify the integrity of binary executable code** – This service is called by the Windows kernel to verify the integrity of digitally signed drivers and other binary components of the operating system.
2. **Show Status** – The module does not provide an explicit status interface. Operational status is indicated by successfully initializing the module using CiInitialize and success status messages using the binary integrity verification functions.
3. **Self-Tests** - The module provides a power-up self-tests service that is automatically executed when the module is loaded into memory.

The following table maps the services to their corresponding algorithms and critical security parameters (CSPs) as described in Cryptographic Key Management.

Table 3 Services

Service / Function	Algorithms	CSPs	Invocation
Verify the integrity of binary executable code	FIPS 186-4 RSA PKCS#1 (v1.5) verify with public key FIPS 180-4 SHS: SHA-1 hash SHA-256 hash SHA-384 hash SHA-512 hash	RSA public key	This service is fully automatic. This service is executed whenever a binary executable is loaded.
Show Status	None	None	This service is fully automatic. This service is executed upon completion of an integrity check function.
Self-Tests	FIPS 186-4 RSA PKCS#1 (v1.5) verify with public key and known signature FIPS 180-4 SHS: SHA-1 KAT SHA-256 KAT SHA-512 KAT	None	This service is fully automatic.

The following table maps services to the export functions listed in [Code Integrity export functions and Code Integrity Callback Functions](#).

Service	Export Functions
Verify the integrity of binary executable code	CiGetPEInformation() CiVerifyHashInCatalog() CiCheckSignedFile() CiFindPageHashesInCatalog() CiFindPageHashesInSignedFile() CiFreePolicyInfo() CiSetTrustedOriginClaimId() CiValidateFileObject() CiValidateImageHeader() CiValidateImageData() CiSetFileCache() CiGetFileCache() CiHashMemory() KappxIsPackageFile() CiCompareSigningLevels()

	CiValidateFileAsImageType() CiRegisterSigningInformation() CiUnregisterSigningInformation() CiValidateDynamicCodePages() SiPolicyQuerySecurityPolicy() CiSetUnlockInformation() CiGetBuildExpiryTime() CiGetStrongImageReference() CiReleaseContext() CiHvciSetImageBaseAddress()
Show Status	CiQueryInformation() CiInitialize() CiInitializePolicy() CipQueryPolicyInformation() CiRevalidateImage() All exported functions
Self-Tests	CiInitialize()

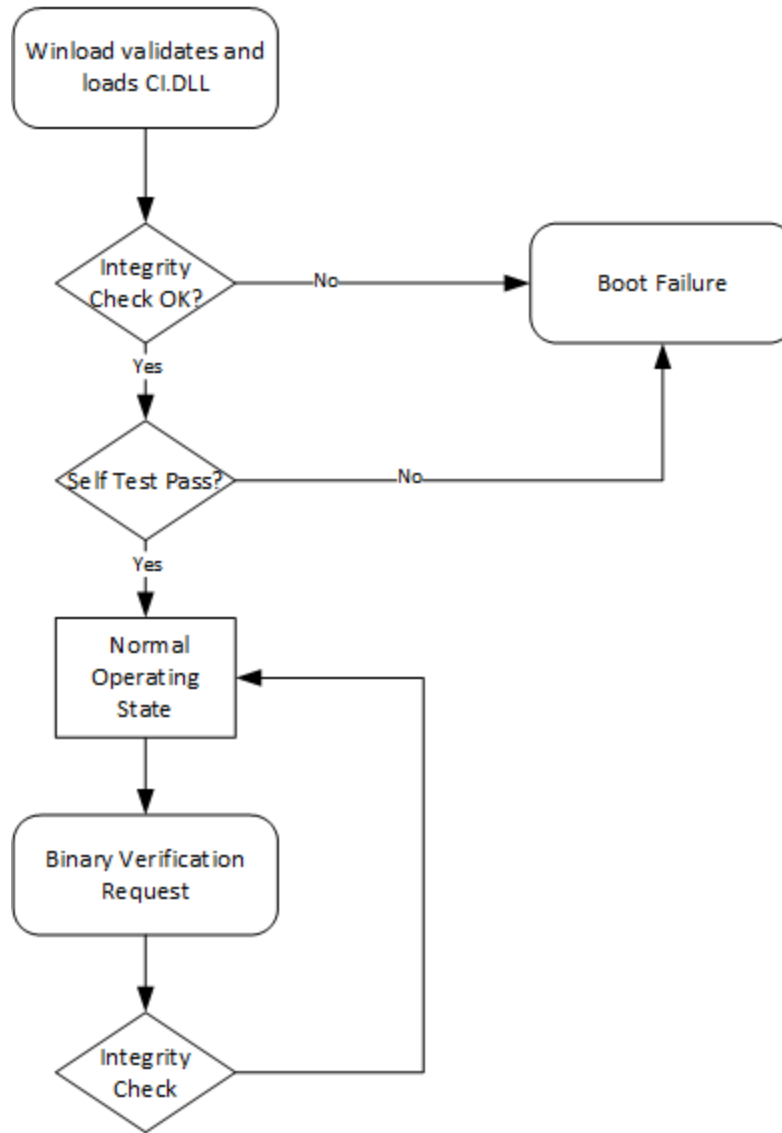
4.3 Authentication

The module does not provide authentication. Roles are implicitly assumed based on the services that are executed.

5 Finite State Model

5.1 Specification

The following diagram shows the finite state model for Code Integrity:



6 Operational Environment

The operational environment for Code Integrity is the Windows Server operating system running on a supported hardware platform listed in section 1.2.

6.1 Single Operator

Code Integrity is invoked by the Windows kernel as a fully automatic service with no user interaction.

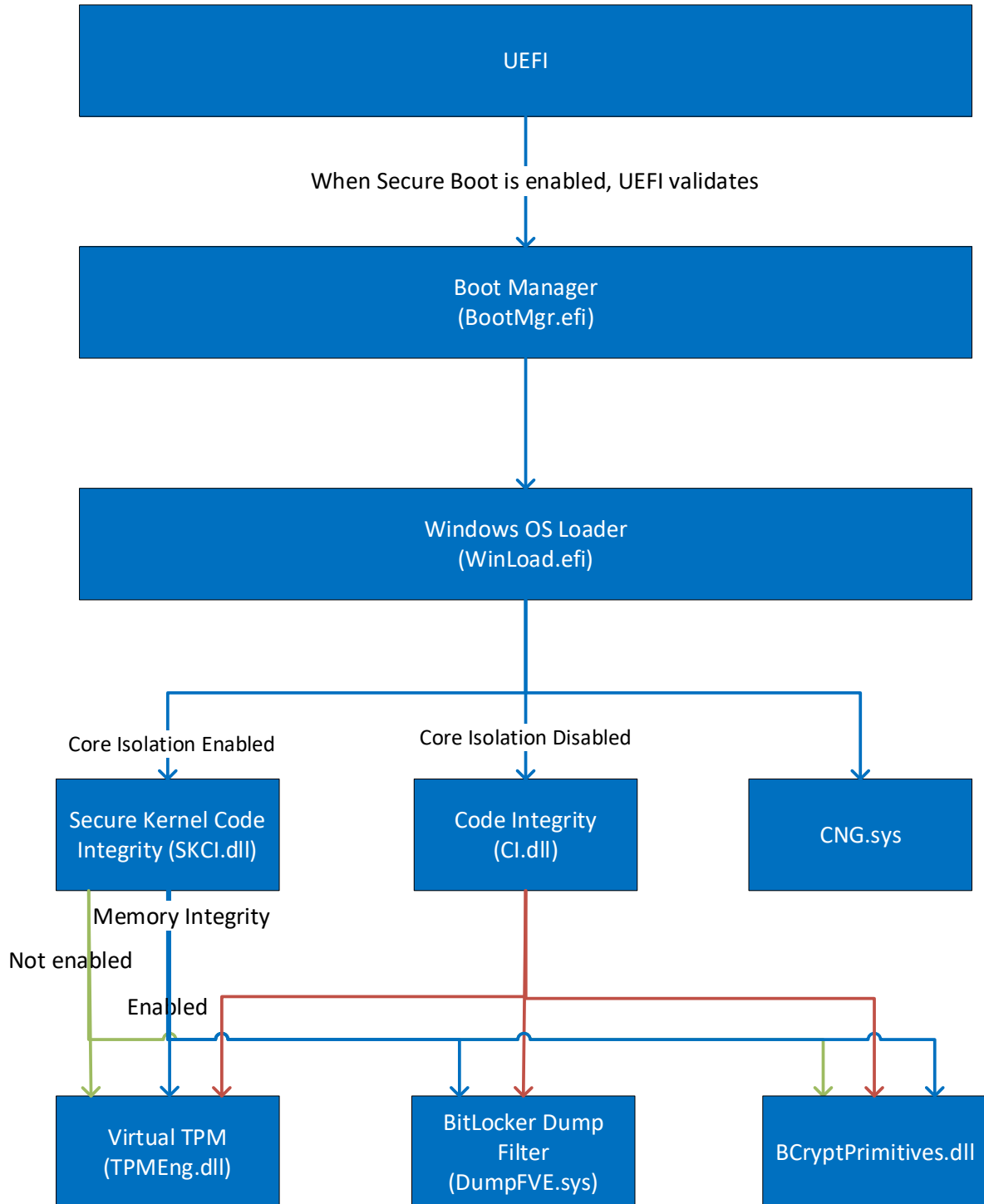
6.2 Cryptographic Isolation

In the Windows operating system, all kernel-mode modules, including CI.DLL, are loaded into the Windows Kernel (ntoskrnl.exe) which executes as a single process. The Windows operating system environment enforces process isolation from user-mode processes including memory and processor scheduling between the kernel and user-mode processes.

6.3 Integrity Chain of Trust

Windows uses several mechanisms to provide integrity verification depending on the stage in the OS boot sequence and also on the hardware and OS configuration. The following diagram describes the Integrity Chain of trust for each supported configuration for the following versions:

- Windows Server 2019 build 10.0.17763.10021 and 10.0.17763.10127



The integrity of the Code Integrity module is checked by the Windows Loader. The Code Integrity module is then, in certain configurations, responsible for checking integrity of every other user and kernel mode system binary as they are loaded.

Refer back to the [introduction](#) for information on the relationship between Code Integrity and Secure Kernel Code Integrity and the effect of configuration on module validation.

7 Cryptographic Key Management

Code Integrity does not generate or store any persistent cryptographic keys; and uses the following RSA public key for validating file integrity:

- Microsoft Root Certificate Authority (CA) Public Key – 2048-bit RSA key with SHA-256.

8 Self-Tests

The Code Integrity module implements Known Answer Test (KAT) functions each time the module is loaded by the Windows kernel and CiInitialize is called.

The module performs 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 Signature of the PKCS#1 v1.5 format with both 1024-bit keys with SHA1 digest and 2048-bit keys with SHA-256 digest.

If any self-test fails, the module will not load and a failure status, STATUS_INVALID_IMAGE_HASH, is returned and the computer will fail to boot. Otherwise STATUS_SUCCESS is returned and the boot process completes.

9 Design Assurance

The secure installation, generation, and startup procedures of this cryptographic module are part of the overall operating system secure installation, configuration, and startup procedures for Windows Server operating system.

The Windows Server operating system must be pre-installed on a computer by an OEM, installed by the end-user, by an organization's IT administrator, or updated from a previous Windows Server version downloaded from Windows Update.

An inspection of authenticity of the physical medium can be made by following the guidance at this Microsoft web site: <https://www.microsoft.com/en-us/howtotell/default.aspx>

The installed version of Windows 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.

10 Mitigation of Other Attacks

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

Table 4 Mitigation of Other Attacks

Algorithm	Protected Against	Mitigation
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

11 Security Levels

The security level for each FIPS 140-2 security requirement is given in the following table.

Table 5 Security Levels

Security Requirement	Security Level
Cryptographic Module Specification	1
Cryptographic Module Ports and Interfaces	1
Roles, Services, and Authentication	1
Finite State Model	1
Physical Security	NA
Operational Environment	1
Cryptographic Key Management	1
EMI/EMC	1
Self-Tests	1

Design Assurance	2
Mitigation of Other Attacks	1

12 Additional Details

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

<https://www.microsoft.com/en-us/windows>

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

<https://docs.microsoft.com/en-us/windows/security/threat-protection/fips-140-validation>

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

13.1 How to Verify Windows Versions

The installed version of Windows must be verified to match the version that was validated using the following method:

1. In the Search box type "cmd" and open the Command Prompt desktop app.
2. The command window will open.
3. At the prompt, enter "ver".
4. The version information will be displayed in a format like this:
`Microsoft Windows [Version 10.0.xxxxx]`

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

13.2 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: xx.x.xxxxx.xxxx.
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.