

# TPM Main

## Part 1 Design Principles

Specification Version 1.2  
Revision 116  
1 March 2011  
TCG Published

Contact: [admin@trustedcomputinggroup.com](mailto:admin@trustedcomputinggroup.com)

## TCG Published

Copyright © 2003-2011 Trusted Computing Group, Incorporated

TCG

33Copyright © 2003-2011 Trusted Computing Group, Incorporated.

#### 34**Disclaimers, Notices, and License Terms**

35THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER,  
36INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR  
37ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY  
38PROPOSAL, SPECIFICATION OR SAMPLE.

39Without limitation, TCG disclaims all liability, including liability for infringement of any  
40proprietary rights, relating to use of information in this specification and to the  
41implementation of this specification, and TCG disclaims all liability for cost of procurement  
42of substitute goods or services, lost profits, loss of use, loss of data or any incidental,  
43consequential, direct, indirect, or special damages, whether under contract, tort, warranty  
44or otherwise, arising in any way out of use or reliance upon this specification or any  
45information herein.

46This document is copyrighted by Trusted Computing Group (TCG), and no license, express or implied, is  
47granted herein other than as follows: You may not copy or reproduce the document or distribute it to others  
48without written permission from TCG, except that you may freely do so for the purposes of (a) examining or  
49implementing TCG specifications or (b) developing, testing, or promoting information technology standards and  
50best practices, so long as you distribute the document with these disclaimers, notices, and license terms.

51

52Contact the Trusted Computing Group at  
53<http://www.trustedcomputinggroup.org/> for information  
54on specification licensing through membership agreements.

55Any marks and brands contained herein are the property of their respective owners.

56

## 57Change History

58

Version	Date	Description
Rev 50	Jun 2003	Started 30 Jun 2003 by David Grawrock First cut at the design principles
Rev 52	Jul 2003	Started 15 Jul 2003 by David Grawrock Moved
Rev 58	Aug 2003	Started 27 Aug 2003 by David Grawrock All emails through 28 August 2003 New delegation from Graeme merged
Rev 62	Oct 2003	Approved by WG, TC and Board as public release of 1.2
Rev 63	Oct 2003	Started 2 Oct 2003 by David Grawrock Kerry email 7 Oct "Various items in rev62" kerry email 10 Oct "Other issues in rev 62" Changes to audit generation
Rev 64	Oct 2003	Started 12 Oct 2003 by David Grawrock Removed PCRWRITE usage in the NV write commands Added locality to transport_out log Disable readpubek now set in takeownership. DisableReadpubek now deprecated, as the functionality is moot. Oshrats email regarding DSAP/OSAP sessions and the invalidation of them on delegation changes Changes for CMK commands. Oshrats email with minor 63 comments
Rev 65	Nov 2003	Action in NV_DefineSpace to ignore the Booleans in the input structure (Kerry email of 10/30 Transport changes from markus 11/6 email Set rules for encryption of parameters for OIAP, OSAP and DSAP Rewrote section on debug PCR to specify that the platform spec must indicate which register is the debug PCR Orlando FtF decisions CMK changes from Graeme
Rev 66	Nov 2003	Comment that OSAP tied to owner delegation needs to be treated internally in the TPM as a DSAP session Minor edits from Monty Added new GetCapability as requested by PC Specific WG Added new DP section that shows mandatory and optional Oshrat email of 11/27 Change PCR attributes to use locality selection instead of an array of BOOL's Removed transport sessions as something to invalidate when a resource type is flushed. Oshrat email of 12/3 added checks for NV_Locked in the NV commands Additional emails from the WG for minor editing fixes
Rev 67	Dec 2003	Made locality_modifier always a 1 size Changed NV index values to add the reserved bit. Also noticed that the previous NV index values were 10 bytes not 8. Edited them to correct size. Audit changes to ensure audit listed as optional and the previous commands properly deleted Added new OSAP authorization encryption. Changes made with new entity types, new section in DP (bottom of doc) and all command rewritten to check for the new encryption
Rev 68	Jan 2004	Added new section to identify all changes made for FIPS. Made some FIPS changes on creating and loading of keys Added change that OSAP encryption IV creation always uses both odd and even nonces Added SEALX ordinal and changes to TPM_STORED_DATA12 and seal/unseal to support this

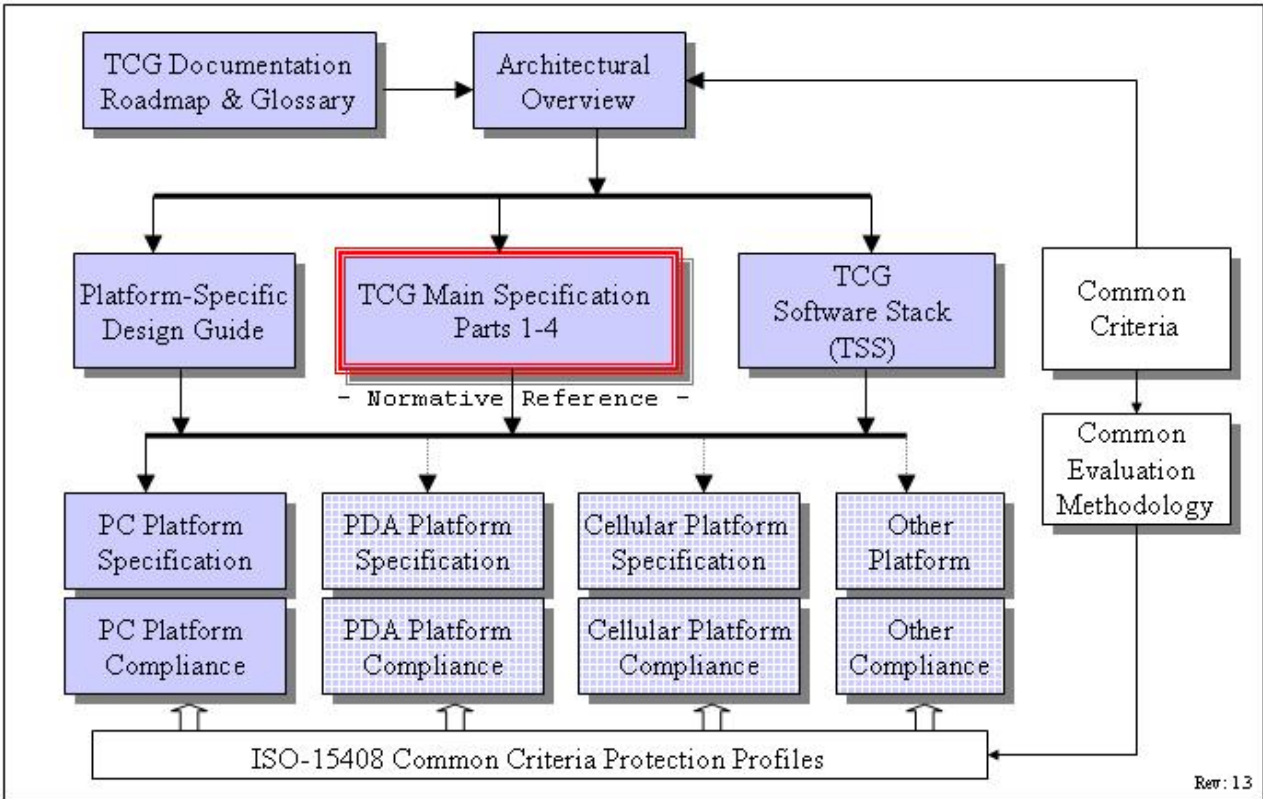
Rev 69	Feb 2004	Fixup on stored_data12. Removed magic4 from the GPIO Added in section 34 of DP further discussion of versioning and getcap DP todo section cleaned up Changed store_privkey in migrate_asymkey Moved text for getcapabilities – hopefully it is easier to read and follow through on now.
Rev 70	Mar 2004	Rewrite structure doc on PCR selection usage. New getcap to answer questions regarding TPM support for pcr selection size
Rev 71	Mar 2004	Change terms from authorization data to AuthData.
Rev 72	Mar 2004	Zimmermann's changes for DAA Added TPM_Quote2, this includes new structure and ordinal Updated key usage table to include the 1.2 commands Added security properties section that links the main spec to the conformance WG guidelines (in section 1)
Rev 73	Apr 2004	Changed CMK_MigrateKey to use TPM_KEY12 and removed two input parameters Allowed TPM_Getcapability and TPM_GetTestResult to execute prior to TPM_Startup when in failure mode
Rev 74	May 2004	Minor editing to reflect comments on web site. Locked spec and submitted for IP review
Rev 76	Aug 2004	All comments from the WG Included new SetValue command and all of the indexes to make that work
Rev 77	Aug 2004	All comments from the WG
Rev 78	Oct 2004	Comments from WG. Added new getcaps to report and query current TPM version
Rev 82	Jan 2005	All changes from emails and minutes (I think).
Rev 84	Feb 2005	Final changes for 1.2 level 2
Rev 88	Aug 2005	Eratta level 2 release candidate
Rev 91	Sept. 2005	Update to Figure 9 (b) in section 9.2 by Tasneem Brutch
Rev 100	May 2006	Clarified CTR mode
Rev 101	Aug 2006	Added deactivated rationale. Clarified number of sessions. Changed "set to NULL" to "set to zero". Added NV index D bit rationale. Added _INFO key rationale and clarified cases where _INFO keys act as _SHA1 keys.
Rev 102	Sept 2006	Minor typos only. No functional changes.
Rev 103	Oct 2006	Note that blobs encrypted in blocks must have integrity chaining. Merged two AIK sections. Self-test checks EK using encryption, not signing.
Rev 104	Nov 2006	RNG must be NIST approved in FIPS mode. When ordinal says OSAP for ADIP, it means OSAP or DSAP.
Rev 105	Feb 2007	Removed informative reference to getting signed counter values.
Rev 106	April 2007	Updated TPM_TakeOwnership state diagram and text to remove deactivated. Indicated that _INFO ordinals are examples, not a complete list.
Rev 107	July 2007	Explained that PCRs are still extended when disabled and deactivated. Removed restriction that TPM_CreateEndorsementKeyPair is not allowed after TPM_CreateRevokableEK or TPM_RevokeTrust.
Rev 108	Aug 2007	Field upgrade should not affect shielded locations. TPM MUST support two key slots.
Rev 109	Oct 2007	Cleaned up Opt-in physical presence wording. Changed some physical presence terms to agree with Part 2.
Rev 110	May 2008	TPM_AUTH_PRIV_USE_ONLY name change and indication that it refers to reading the public key. Warning that ADIP with a well known secret may require transport.
Rev 111	July 2008	TSC_ ordinals must be tested early. AddedMUST to self tests, but indicated that test methods are examples. Key contexts cannot be flushed. Deleted unbind payload normative to agree with part 3.
Rev 112	Jan 2009	Default exponent clarified, self-test failure cleared by TPM_Init, self-test failure must delete saved state
Rev 113	Jan 2009	No changes.
Rev 114	Jan 2009	No changes

20

Rev 116	Aug 2009	Audit happens at the end of the command. Audit occurs when deactivated to remove a corner case.
---------	----------	---

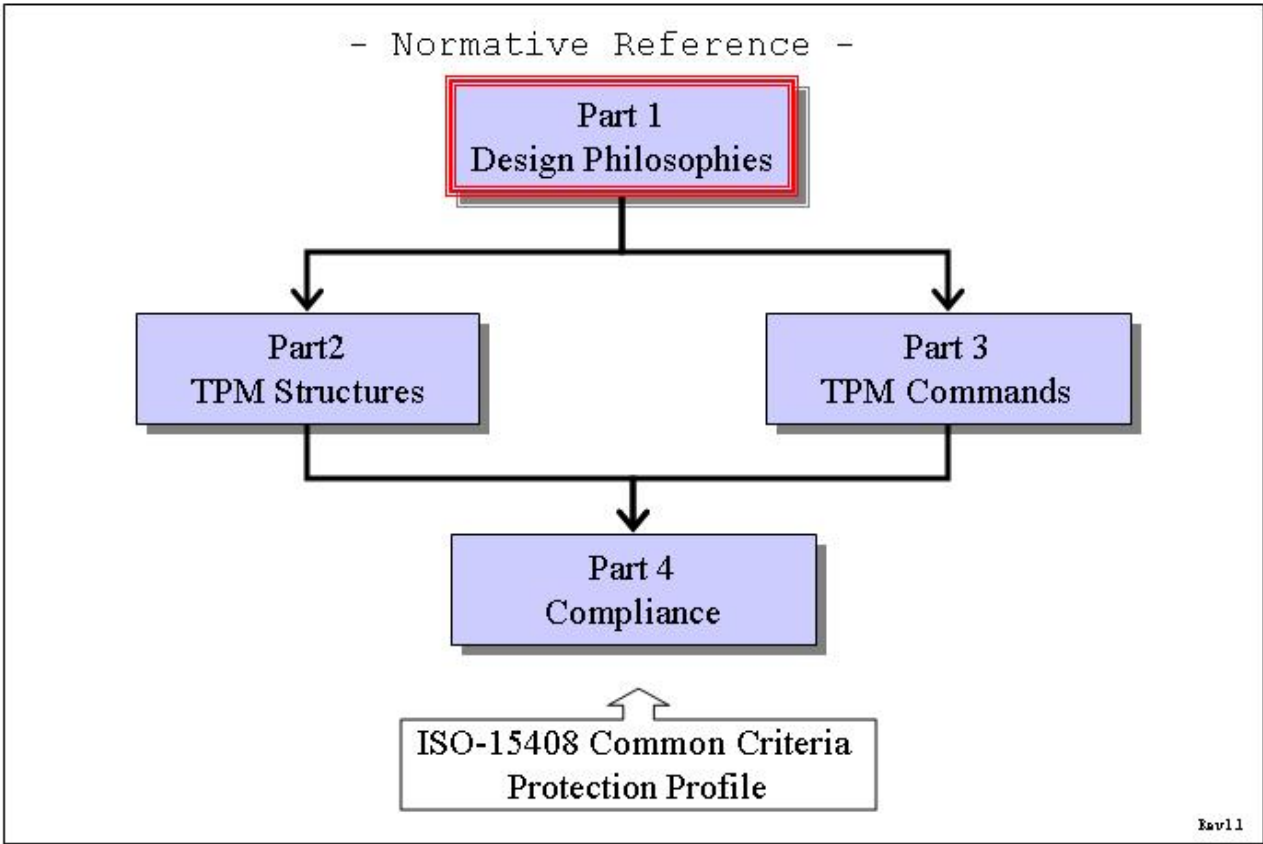
59

# TCG Doc Roadmap – Main Spec



60

# TCG Main Spec Roadmap



61

62**Table of Contents**

631. Scope and Audience..... 1

64 1.1 Key words..... 1

65 1.2 Statement Type..... 1

662. Description..... 2

67 2.1 TODO (notes to keep the editor on track) ..... 2

68 2.2 Questions..... 2

69 2.2.1 Delegation Questions..... 6

70 2.2.2 NV Questions..... 11

713. Protection..... 13

72 3.1 Introduction ..... 13

73 3.2 Threat..... 14

74 3.3 Protection of functions..... 14

75 3.4 Protection of information..... 14

76 3.5 Side effects..... 15

77 3.6 Exceptions and clarifications..... 15

784. TPM Architecture..... 17

79 4.1 Interoperability..... 17

80 4.2 Components..... 17

81 4.2.1 Input and Output..... 18

82 4.2.2 Cryptographic Co-Processor..... 18

83 4.2.2.1 RSA Engine..... 19

84 4.2.2.2 Signature Operations..... 19

85 4.2.2.3 Symmetric Encryption Engine..... 20

86 4.2.2.4 Using Keys..... 20

87 4.2.3 Key Generation..... 21

88 4.2.3.1 Asymmetric – RSA..... 21

89 4.2.3.2 Nonce Creation..... 21

90 4.2.4 HMAC Engine..... 21

91 4.2.5 Random Number Generator..... 22

92 4.2.5.1 Entropy Source and Collector..... 23

93 4.2.5.2 State Register..... 24

94 4.2.5.3 Mixing Function..... 24

95 4.2.5.4 RNG Reset..... 24

96 4.2.6 SHA-1 Engine..... 25

97 4.2.7 Power Detection..... 25



38		
98	<a href="#">4.2.8 Opt-In.....</a>	<a href="#">26</a>
99	<a href="#">4.2.9 Execution Engine.....</a>	<a href="#">27</a>
100	<a href="#">4.2.10 Non-Volatile Memory.....</a>	<a href="#">28</a>
101	<a href="#">4.3 Data Integrity Register (DIR).....</a>	<a href="#">28</a>
102	<a href="#">4.4 Platform Configuration Register (PCR).....</a>	<a href="#">28</a>
103	<a href="#">5. Endorsement Key Creation.....</a>	<a href="#">31</a>
104	<a href="#">5.1 Controlling Access to PRIVEK.....</a>	<a href="#">32</a>
105	<a href="#">5.2 Controlling Access to PUBEK.....</a>	<a href="#">32</a>
106	<a href="#">6. Attestation Identity Keys.....</a>	<a href="#">33</a>
107	<a href="#">7. TPM Ownership.....</a>	<a href="#">34</a>
108	<a href="#">7.1 Platform Ownership and Root of Trust for Storage.....</a>	<a href="#">34</a>
109	<a href="#">8. Authentication and Authorization Data.....</a>	<a href="#">35</a>
110	<a href="#">8.1 Dictionary Attack Considerations.....</a>	<a href="#">36</a>
111	<a href="#">9. TPM Operation.....</a>	<a href="#">38</a>
112	<a href="#">9.1 TPM Initialization &amp; Operation State Flow.....</a>	<a href="#">39</a>
113	<a href="#">9.1.1 Initialization .....</a>	<a href="#">39</a>
114	<a href="#">9.2 Self-Test Modes.....</a>	<a href="#">41</a>
115	<a href="#">9.2.1 Operational Self-Test.....</a>	<a href="#">43</a>
116	<a href="#">9.3 Startup .....</a>	<a href="#">47</a>
117	<a href="#">9.4 Operational Mode.....</a>	<a href="#">47</a>
118	<a href="#">9.4.1 Enabling a TPM.....</a>	<a href="#">48</a>
119	<a href="#">9.4.2 Activating a TPM.....</a>	<a href="#">50</a>
120	<a href="#">9.4.3 Taking TPM Ownership.....</a>	<a href="#">51</a>
121	<a href="#">9.4.3.1 Enabling Ownership.....</a>	<a href="#">52</a>
122	<a href="#">9.4.4 Transitioning Between Operational States .....</a>	<a href="#">53</a>
123	<a href="#">9.5 Clearing the TPM .....</a>	<a href="#">54</a>
124	<a href="#">10. Physical Presence .....</a>	<a href="#">56</a>
125	<a href="#">11. Root of Trust for Reporting (RTR).....</a>	<a href="#">58</a>
126	<a href="#">11.1 Platform Identity.....</a>	<a href="#">58</a>
127	<a href="#">11.2 RTR to Platform Binding .....</a>	<a href="#">59</a>
128	<a href="#">11.3 Platform Identity and Privacy Considerations.....</a>	<a href="#">59</a>
129	<a href="#">11.4 Attestation Identity Keys.....</a>	<a href="#">59</a>
130	<a href="#">11.4.1 AIK Creation.....</a>	<a href="#">60</a>
131	<a href="#">11.4.2 AIK Storage.....</a>	<a href="#">61</a>
132	<a href="#">12. Root of Trust for Storage (RTS).....</a>	<a href="#">62</a>
133	<a href="#">12.1 Loading and Unloading Blobs.....</a>	<a href="#">62</a>
134	<a href="#">13. Transport Sessions and Authorization Protocols.....</a>	<a href="#">63</a>

135	<a href="#">13.1 Authorization Session Setup.....</a>	<a href="#">66</a>
136	<a href="#">13.2 Parameter Declarations for OIAP and OSAP Examples.....</a>	<a href="#">67</a>
137	<a href="#">13.2.1 Object-Independent Authorization Protocol (OIAP) .....</a>	<a href="#">70</a>
138	<a href="#">13.2.2 Object-Specific Authorization Protocol (OSAP) .....</a>	<a href="#">74</a>
139	<a href="#">13.3 Authorization Session Handles.....</a>	<a href="#">78</a>
140	<a href="#">13.4 Authorization-Data Insertion Protocol (ADIP).....</a>	<a href="#">79</a>
141	<a href="#">13.5 AuthData Change Protocol (ADCP) .....</a>	<a href="#">83</a>
142	<a href="#">13.6 Asymmetric Authorization Change Protocol (AACP).....</a>	<a href="#">84</a>
143	<a href="#">14. FIPS 140 Physical Protection.....</a>	<a href="#">85</a>
144	<a href="#">14.1 TPM Profile for FIPS Certification.....</a>	<a href="#">85</a>
145	<a href="#">15. Maintenance.....</a>	<a href="#">86</a>
146	<a href="#">15.1 Field Upgrade.....</a>	<a href="#">87</a>
147	<a href="#">16. Proof of Locality.....</a>	<a href="#">89</a>
148	<a href="#">17. Monotonic Counter .....</a>	<a href="#">90</a>
149	<a href="#">18. Transport Protection.....</a>	<a href="#">93</a>
150	<a href="#">18.1 Transport encryption and authorization.....</a>	<a href="#">95</a>
151	<a href="#">18.1.1 MGF1 parameters.....</a>	<a href="#">97</a>
152	<a href="#">18.1.2 HMAC calculation.....</a>	<a href="#">97</a>
153	<a href="#">18.1.3 Transport log creation.....</a>	<a href="#">98</a>
154	<a href="#">18.1.4 Additional Encryption Mechanisms.....</a>	<a href="#">98</a>
155	<a href="#">18.2 Transport Error Handling.....</a>	<a href="#">98</a>
156	<a href="#">18.3 Exclusive Transport Sessions.....</a>	<a href="#">99</a>
157	<a href="#">18.4 Transport Audit Handling.....</a>	<a href="#">100</a>
158	<a href="#">18.4.1 Auditing of wrapped commands.....</a>	<a href="#">100</a>
159	<a href="#">19. Audit Commands.....</a>	<a href="#">101</a>
160	<a href="#">19.1 Audit Monotonic Counter.....</a>	<a href="#">103</a>
161	<a href="#">20. Design Section on Time Stamping.....</a>	<a href="#">104</a>
162	<a href="#">20.1 Tick Components.....</a>	<a href="#">104</a>
163	<a href="#">20.2 Basic Tick Stamp.....</a>	<a href="#">105</a>
164	<a href="#">20.3 Associating a TCV with UTC.....</a>	<a href="#">105</a>
165	<a href="#">20.4 Additional Comments and Questions.....</a>	<a href="#">107</a>
166	<a href="#">21. Context Management.....</a>	<a href="#">110</a>
167	<a href="#">22. Eviction.....</a>	<a href="#">112</a>
168	<a href="#">23. Session pool.....</a>	<a href="#">113</a>
169	<a href="#">24. Initialization Operations .....</a>	<a href="#">114</a>
170	<a href="#">25. HMAC digest rules.....</a>	<a href="#">116</a>
171	<a href="#">26. Generic authorization session termination rules.....</a>	<a href="#">117</a>

172	<a href="#">27. PCR Grand Unification Theory.....</a>	<a href="#">118</a>
173	<a href="#">27.1 Validate Key for use.....</a>	<a href="#">121</a>
174	<a href="#">28. Non Volatile Storage.....</a>	<a href="#">122</a>
175	<a href="#">28.1 NV storage design principles.....</a>	<a href="#">123</a>
176	<a href="#">28.1.1 NV Storage use models.....</a>	<a href="#">124</a>
177	<a href="#">28.2 Use of NV storage during manufacturing.....</a>	<a href="#">125</a>
178	<a href="#">29. Delegation Model.....</a>	<a href="#">127</a>
179	<a href="#">29.1 Table Requirements .....</a>	<a href="#">127</a>
180	<a href="#">29.2 How this works.....</a>	<a href="#">128</a>
181	<a href="#">29.3 Family Table.....</a>	<a href="#">130</a>
182	<a href="#">29.4 Delegate Table.....</a>	<a href="#">131</a>
183	<a href="#">29.5 Delegation Administration Control.....</a>	<a href="#">132</a>
184	<a href="#">29.5.1 Control in Phase 1.....</a>	<a href="#">133</a>
185	<a href="#">29.5.2 Control in Phase 2.....</a>	<a href="#">134</a>
186	<a href="#">29.5.3 Control in Phase 3.....</a>	<a href="#">134</a>
187	<a href="#">29.6 Family Verification.....</a>	<a href="#">134</a>
188	<a href="#">29.7 Use of commands for different states of TPM.....</a>	<a href="#">136</a>
189	<a href="#">29.8 Delegation Authorization Values.....</a>	<a href="#">136</a>
190	<a href="#">29.8.1 Using the authorization value .....</a>	<a href="#">137</a>
191	<a href="#">29.9 DSAP description.....</a>	<a href="#">137</a>
192	<a href="#">30. Physical Presence.....</a>	<a href="#">141</a>
193	<a href="#">30.1 Use of Physical Presence.....</a>	<a href="#">141</a>
194	<a href="#">31. TPM Internal Asymmetric Encryption.....</a>	<a href="#">143</a>
195	<a href="#">31.1.1 TPM_ES_RSAESOAEP_SHA1_MGF1.....</a>	<a href="#">143</a>
196	<a href="#">31.1.2 TPM_ES_RSAESPKCSV15.....</a>	<a href="#">144</a>
197	<a href="#">31.1.3 TPM_ES_SYM_CTR.....</a>	<a href="#">144</a>
198	<a href="#">31.1.4 TPM_ES_SYM_OFB.....</a>	<a href="#">144</a>
199	<a href="#">31.2 TPM Internal Digital Signatures.....</a>	<a href="#">145</a>
200	<a href="#">31.2.1 TPM_SS_RSASSAPKCS1v15_SHA1.....</a>	<a href="#">145</a>
201	<a href="#">31.2.2 TPM_SS_RSASSAPKCS1v15_DER.....</a>	<a href="#">146</a>
202	<a href="#">31.2.3 TPM_SS_RSASSAPKCS1v15_INFO.....</a>	<a href="#">146</a>
203	<a href="#">31.2.4 Use of Signature Schemes.....</a>	<a href="#">146</a>
204	<a href="#">32. Key Usage Table.....</a>	<a href="#">148</a>
205	<a href="#">33. Direct Anonymous Attestation.....</a>	<a href="#">150</a>
206	<a href="#">33.1 TPM_DAA_JOIN.....</a>	<a href="#">150</a>
207	<a href="#">33.2 TPM_DAA_Sign.....</a>	<a href="#">152</a>
208	<a href="#">33.3 DAA Command summary.....</a>	<a href="#">152</a>

209    [33.3.1 TPM setup.....](#) [153](#)

210    [33.3.2 JOIN.....](#) [153](#)

211    [33.3.3 SIGN.....](#) [157](#)

212 [34. General Purpose IO.....](#) [160](#)

213 [35. Redirection.....](#) [161](#)

214 [36. Structure Versioning.....](#) [162](#)

215 [37. Certified Migration Key Type.....](#) [164](#)

216    [37.1 Certified Migration Requirements.....](#) [164](#)

217    [37.2 Key Creation.....](#) [165](#)

218    [37.3 Migrate CMK to a MA.....](#) [165](#)

219    [37.4 Migrate CMK to a MSA.....](#) [166](#)

220 [38. Revoke Trust.....](#) [167](#)

221 [39. Mandatory and Optional Functional Blocks.....](#) [169](#)

222 [40. 1.1a and 1.2 Differences.....](#) [172](#)

223

## 2241. Scope and Audience

225The TPM main specification is an industry specification that enables trust in computing  
226platforms in general. The main specification is broken into parts to make the role of each  
227document clear. A version of the specification (like 1.2) requires all parts to be a complete  
228specification.

229A TPM designer MUST be aware that for a complete definition of all requirements necessary  
230to build a TPM, the designer MUST use the appropriate platform specific specification for all  
231TPM requirements.

### 2321.1 Key words

233The key words “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,”  
234“SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” in the chapters 2-10  
235normative statements are to be interpreted as described in [RFC-2119].

### 2361.2 Statement Type

237Please note a very important distinction between different sections of text throughout this  
238document. You will encounter two distinctive kinds of text: informative comment and  
239normative statements. Because most of the text in this specification will be of the kind  
240normative statements, the authors have informally defined it as the default and, as such,  
241have specifically called out text of the kind informative comment They have done this by  
242flagging the beginning and end of each informative comment and highlighting its text in  
243gray. This means that unless text is specifically marked as of the kind informative  
244comment, you can consider it of the kind normative statements.

245For example:

#### 246Start of informative comment

247This is the first paragraph of several paragraphs containing text of the kind *informative*  
248*comment* ...

249This is the second paragraph of text of the kind *informative comment* ...

250This is the nth paragraph of text of the kind *informative comment* ...

251To understand the TCG specification the user must read the specification. (This use of  
252MUST does not require any action).

#### 253End of informative comment

254This is the first paragraph of one or more paragraphs (and/or sections) containing the text  
255of the kind normative statements ...

256To understand the TCG specification the user MUST read the specification. (This use of  
257MUST indicates a keyword usage and requires an action).

## 2582. Description

259The design principles give the basic concepts of the TPM and generic information relative to  
260TPM functionality.

261A TPM designer MUST review and implement the information in the TPM Main specification  
262(parts 1-4) and review the platform specific document for the intended platform. The  
263platform specific document will contain normative statements that affect the design and  
264implementation of a TPM.

265A TPM designer MUST review and implement the requirements, including testing and  
266evaluation, as set by the TCG Conformance Workgroup. The TPM MUST comply with the  
267requirements and pass any evaluations set by the Conformance Workgroup. The TPM MAY  
268undergo more stringent testing and evaluation.

269The question section keeps track of questions throughout the development of the  
270specification and hence can have information that is no longer current or moot. The  
271purpose of the questions is to track the history of various decisions in the specification to  
272allow those following behind to gain some insight into the committees thinking on various  
273points.

### 2742.1 TODO (notes to keep the editor on track)

275

### 2762.2 Questions

#### 277Start of informative comment

278How to version the flag structures?

279 I suggest that we simply put the version into the structure and pass it back in the  
280 structure. Add the version information into the persistent and volatile flag structures.

281When using the encryption transport failures are easy to see. Also the watcher on the line  
282can tell where the error occurred. If the failure occurs at the transport level the response is  
283an error (small packet) and it is in the clear. If the error occurs during execution of the  
284command then the response is a small encrypted packet. Should we expand the packet size  
285or simply let this go through?

286 Not an issue.

287Do we restrict the loading of a counter to once per TPM\_Startup(Clear)?

288 Yes once a counter is set it must remain the same until the next successful startup.

289Does the time stamp work as a change on the tag or as a wrapped command like the  
290transport protection.

291 While possibly easier at the HW level the tag mechanism seems to be harder at the SW  
292 level as to what commands are sent to the TPM. The issue of how the SW presents the  
293 TS session to the SW writer is not an issue. This is due to the fact that however the  
294 session is presented to the SW writer the writer must take into account which  
295 commands are being time stamped and how to manage the log etc. So accepting a  
296 mechanism that is easy for the HW developer and having the SW manage the interface is  
297 a sufficient direction.

65

298When returning time information do we return the entire time structure or just the time  
299and have the caller obtain all the information with a GetCap call?

300 All time returns will use the entire structure with all the details.

301Do we want to return a real clock value or a value with some additional bits (like a  
302monotonic value with a time value)?

303 Add a count value into the time structure.

304Do we need NTP or is SNTP sufficient?

305 The TPM will not run the time protocol itself. What the TPM will do is accept a value  
306 from outside software and a hash of the protocols that produced the value. This allows  
307 the platform to use whatever they want to set the value from secure time to the local PC  
308 clock.

309Can an owner destroy a TPM by issuing repeated CreateCounter commands?

310 A TPM may place a throttle on this command to avoid burn issues. It MUST not be  
311 possible to burn out the TPM counter under normal operating conditions. The  
312 CreateCounter command is limited to only once per successful  
313 TPM\_Startup(ST\_CLEAR).

314 This answer is now somewhat moot as the command to createcounter is now owner  
315 authorized. This allows the owner to decide when to authorize the counter creation. As  
316 there are only 4 counters available it is not an issue with having the owner continue to  
317 authorize counters.

318What happens to a transport session (log etc.) on an S3?

319 Should these be the same as the authorization sessions? The saving of a transport  
320 session across S3 is not a security concern but is a memory concern. The TPM MUST  
321 clear the transport session on TPM\_Startup(CLEAR) and MAY clear the session on  
322 TPM\_Startup(any).

323While you can't increment or create a new counter after startup can you read a counter  
324other than the active one?

325 You may read other counters

326When we audit a command that is not authorized should we hash the parameters and  
327provide that as part of the audit event, currently they are set to null.

328 We should hash parameters of non-authorized commands

329There is a fundamental problem with the encryption of commands in the transport and  
330auditing. If we cover a command we have no way to audit, if we show the command then it  
331isn't protected. Can we expose the command (ordinal) and not the parameters?

332 If the owner has requested that a function be audited then the execute transport return  
333 will include sufficient information to produce the audit entry.

334How to set the time in the audit structure and tell the log what is going on.

335 The time in the audit structure is set to nulls except when audit occurs as part of a  
336 transport session. In that case the audit command is set from the time value in the TPM.

337Is there a limit to the number of locality modifiers?

338 Yes, the TPM need only support a maximum of 4 modifiers. The definition of the  
339 modifiers is always a platform specific issue.

340How do we evict various resources?

341 There are numerous eviction routines in the current spec. We will deprecate the various  
342 types and move to TPM\_Flushxxx for all resource types.

343Can you flush a saved context?

344 Yes, you must be able to invalidate saved contexts. This would be done by making sure  
345 that the TPM could not load any saved context.

346What is the value of maintaining the clock value when the time is not incrementing? Can  
347this be due to the fact that the time is now known to be at least after the indicated time?

348 Moot point now as we don't keep the clock value at

349Should we change the current structures and add the tag?

350 TODO

351Can we have a bank of bits (change bit locality) for each of the 4 levels of locality?

352 Now

353How do we find out what sessions are active? Do we care?

354 I would say yes we care and we should use the same mechanism that we do for the keys.  
355 A GetCap that will return the handles.

356Can we limit the transport sessions to only one?

357 No, we should have as a minimum 2 sessions. One gets into deadlocks and such so the  
358 minimum should be 2.

359 Changed: The deadlock is with authorization sessions, not transport.

360Does the TPM need to keep the audit structure or can it simply keep a hash?

361 The TPM just keeps the audit digest and no other information.

362What happens to an OSAP session if the key associated with it is taken off chip with a  
363"SaveContext"? What happens if the key saveContext occurs after an OSAP auth context  
364that is already off chip? How do you later connect the key to the auth session (without  
365having to store all sorts of things on chip)? Are we really honestly convinced that we've  
366thought of all the possible ramifications of saving and restoring auth sessions? And is it  
367really true that all the things we say about a saved auth session do/should apply to a saved  
368key (which is to say is there really a single loadContext command and a single context  
369structure)?

370 Saved context a reliable indication of the linkage between the OSAP and the key. When  
371 saving save auth then key, on load key then auth. Auth session checks for the key and if  
372 not found fails.

373Why is addNonce an output of 16.5 loadContext?

374 If it's wrong, it's a little late to find out now - why not have it as an input and have the  
375 TPM return an error if the encrypted addNonce doesn't match the input? The thought  
376 was that the nonce area might not be a nonce but was information that the caller could



377 put in. If they use it as a nonce fine, but they could also use it as a label or sequence  
378 number or ... any value the caller wanted

379Is there a memory endurance problem with contextNonceSession?

380 contextNonceSession does not have to be saved across S3 states so there is no  
381 endurance problem.

382Is there a memory endurance problem with contextNonceKey?

383 contextNonceKey only changes on TPM\_Startup(ST\_Clear) so it's endurance is the same  
384 as a PCR.

385The debate continues about restoring a resource's handle during TPM\_LoadContext.

386 Debate ends by having the load context be informed of what the loaders opinion is about  
387 the handle. The requestor can indicate that it wishes the same handle and if the TPM  
388 can perform that task it does, if it cannot then the load fails.

389Interesting attack is now available with the new audit close flag on get audit signed. Anyone  
390with access to a signing key can close the audit log. The only requirement on the command  
391is that the key be authorized. While there is no loss of information (as the attacker can  
392always destroy the external log) does the closing of a log make things look different. This  
393does enable a burn out attack. The ability to closeAudit enables a new DenialOfService  
394attack.

395 Resolution: The TPM Owner owns the audit process, so the TPM Owner should have  
396 exclusive control over closeAudit. Hence the signing key used to closeAudit must be an  
397 AIK. Note that the owner can choose to give this AIK's AuthData value to the OS, so that  
398 the OS can automatically close an audit session during platform power down. But such  
399 operations are outside this specification.

400Should we keep the E function in the tick counter?

401 From Graeme, I would prefer to see these calculations deleted. The calculation starts  
402 with one assertion and derives a contradictory assertion. Generally, there seems little  
403 value in trying to derive an equality relationship when nothing is known about the path  
404 to and from the Time Authority.

405What is the difference between DIR\_Quote and DirReadSigned?

406 Appears to be none so DIR\_Quote deleted

407The tickRate parameter associates tick with seconds and has no way to indicate that the  
408rate is greater than one second. Is this OK?

409 Do we need to allow for tick rates that are slower than once per second. We report in  
410 nanoseconds.

411The TPM MUST support a minimum of 2 authorization sessions. Where do we put this  
412requirement in the spec?

413Can we find a use for the DIR and BIT areas for locality 0?

414 They have no protections so in many ways they are just extra. We leave this as it is as  
415 locality 0 may mean something else on a platform other than a PC.

416How do we send back the transport log information on each execute transport?

417 It is 64 bytes in length and would make things very difficult to include on every  
418 command. Change wrappedaudit to be input params, add output params and the caller  
419 has all information necessary to create the structure to add into the digest.

420The transport log structure is a single structure used both for input and output with the  
421only difference being the setting of ticks to 0 on input and a real value on output, do we  
422need two structures.

423 I believe that a single structure is fine

424For TPM\_Startup(ST\_Clear) I added that all keys would be flushed. Is this right?

425 Yes

426Why have 2 auths for release transport signed? It is an easy attack to simply kill the  
427session.

428 The reason is that an attacker can close the session and get a signature of the session  
429 log. We are currently not sure of the level of this attack but by having the creator of the  
430 session authorize the signing of the log it is completely avoided.

43119.3 Action 3 (startup/state) doesn't reference the situation where there is no saved state.  
432My presumption is that you can still run startup/clear, but maybe you have to do a  
433hardware reset?

434 DWG I don't think so. This could be an attack and a way to get the wrong PCR values  
435 into the system. The BIOS is taking one path and may not set PCR values. Hence the  
436 response is to go into failed selftest mode.

437What happens to a transport session if a command clears the TPM like revokeTrust

438 This is fine. The transport session is not complete but the session protected the  
439 information till the command that changed the TPM. It is impossible to get a log from  
440 the session or to sign the session but that is what the caller wanted.

441**End of informative comment**

## 442**2.2.1 Delegation Questions**

443**Start of informative comment**

444Is loading the table by untrusted process ok? Does this cause a problem when the new table  
445is loaded and permissions change?

446 Yes, the fill table can be done by any process. A TPM Owner wishing to validate the table  
447 can perform the operations necessary to gain assurance of the table entries.

448Are the permissions for a table row sensitive?

449 Currently we believe not but there are some attack models that knowing the permissions  
450 makes the start of the attack easier. It does not make the success of the attack any  
451 easier. Example if I know that a single process is the only process in the table that has  
452 the CreateAIK capability then the attacker only attempts to break into the single process  
453 and not all others.

454What software is in use to modify the table?

83

455 The table can be updated by any software or process given the capability to manage the  
456 table. Three likely sources of the software would be a BIOS process, an applet of a  
457 trusted process and a standalone self-booting (from CD-ROM) management application.

458Who holds the TPM Owner password?

459 There is no change to the holding of the TPM Owner token. The permissions do allow the  
460 creation of an application that sets the TPM Owner token to a random value and then  
461 seals the value to the application.

462How are these changes created such that there is minimal change to the current TPM?

463 This works by using the current authorization process and only making changes in the  
464 authorization and not for each and every command.

465What about S3 and other events?

466 Permissions, once granted, are non-volatile.

467The permission bit to changeOwnerAuth (bit 11) gives rise to the functionality that the SW  
468that has this bit can control the TPM completely. This includes removing control from the  
469TPM Owner as the TPM Owner value will now be a random value only known to SW. There  
470are use models where this is good and bad, do we want this functionality?

471Pros and cons of physical enable table when TPM Owner is present – Pro physically present  
472user can make SW play fair. Con – physically present user can override the desires of a TPM  
473Owner.

474Do we need to reset TPM\_PERMISSION\_KEY at some time?

475 We know that the key is NOT reset on TPM\_ClearOwner.

476What is the meaning of using permission table in an OIAP and OSAP mode?

477 Delegate table can be used in either OIAP or OSAP mode.

478Can you grant permissions without assigning the permissions to a specific process?

479 Yes, do a SetRow with a PCR\_SELECTION of null and the permissions are available to  
480 any process.

481Do we need a ClearTableOwner?

482 I would assert that we do not need this command. The TPM Owner can perform SetRow  
483 with NULLS four times and creates the exact same thing. Not having this command  
484 lowers the number of ordinals the TPM is required to support.

485There are some issues with the currently defined behavior of familyID and the  
486verificationCount.

487 Talked to David for 30 mins. We decided that maxFamilyID is set to zero at  
488 manufacture, and incremented for every FamTable\_SetRow

489 It is the responsibility of DelTable\_SetRow to set the appropriate familyID

490 DelTable\_SetRow fails if the provided familyID is not active and present somewhere in  
491 the FamTable

492 FillTable works differently. It effectively resets the family table (invalidating all active  
493 rows) and sets up as many rows as are needed based on the number of families specified  
494 in FillTable

495 This still needs a bit of work. Presumably the caller of FillTable uses a “fake” familyID,  
496 and this is changed to the actual familyID when the fill happens

497 There are some issues with the verificationCount.

498 Uber-issue. If none of the rows in the table are allowed to create other rows and export  
499 them, then the “sign” of the table is meaningful

500 If one of the rows is allowed to create and export new rows, is there any real meaning to  
501 “the current set of exported rows?” (i.e. SW can just up and make new rows).

502 Should section 4.4, TPM\_DelTable\_ClearTable), section 4.5 (TPM\_DelTable\_SetEnable), and  
503 section 4.7 (TPM\_DelTable\_Set\_Admin) all say “there must be UNAMBIGUOUS evidence of  
504 the presence of physical access...” Is this okay?

505 Answer: No, group agreed to change UNAMBIGUOUS to BEST EFFORT in all three  
506 sections.

507 Is FamilyID a sensitive value?

508 If so, why? Agreement: FamilyID is not a sensitive value.

509 Should TPM\_TakeOwnership be included in permissions bits (see bit 12 in section 3.1)?

510 Enables a better administrative monitor and may enable user to take ownership easier.  
511 Agreement leave it in and change informative comments to reflect the reasons.

512 [From the TPM\_DelTable\_SetRow command informative comments]: Note that there are two  
513 types of rights: family rights (you can either edit your family’s rows or grab new rows) and  
514 administrative rights.

515 This is really just an editor’s note, not a question to be resolved.

516 [From the TPM\_DelTable\_ExportRow command informational comments]:

517 Does not effect content of exported row left behind in the table;

518 Valid for all rows in the table;

519 Does not need to be OwnerAuth’d;

520 Family Rights are that family can only export a row from rows 0-3 if row belongs to the  
521 family, but rows 4 and upwards can be exported by any Trusted Process, without any  
522 family checking being done. This is really just an editor’s note, not a question to be  
523 resolved.

524 When a Family Table row is set, the verificationCount is set to 1, make sure that is  
525 consistently used in all other command actions.

526 Done.

527 SetEnable and SetEnableOwner enable and disable all rows in a table, not just the rows  
528 belong to the family of the process that used the SetEnable and/or SetEnableOwner  
529 commands. This is also true for SetAdmin and SetAdminOwner. Can anybody come up with  
530 a use scenario where that causes any problems?

531 In command actions where the TPM must walk the delegation table looking for a  
532 configuration that matches the command input parameters (PCRinfo and/or authValues)  
533 and there are rows in the table with duplicate values, what does the TPM do? Is there any

92

534reason not to use the rule “the TPM starts walking the table starting with the first row and  
535use the first row it finds with matching values”?

536 Answer to this question may mean change to pseudo code in section 2.3, Using the  
537 AuthData Value, which currently shows the TPM walking the delegation table, starting  
538 with the first row, and using the first row it finds with matching values.

539What familyID value signals a family table row that is not in use/contains invalid values?

540 To get consistency in all the command Actions that use this, that FamilyID value has  
541 been edited in all places to be NULL, instead of 0. Yes, FamilyID value of NULL signals a  
542 family table row that is not in use or contains invalid values.

543From section 2.4, Delegate Table Fill and Enablement: “The changing of a TPM Owner does  
544not automatically clear the delegate table. Changing a TPM Owner does disable all current  
545delegations, including exported rows, and requires the new TPM Owner to re-enable the  
546delegations in the table. The table entry values like trusted process identification and  
547delegations to that process are not effected by a change in owner. THE AUTHDATA VALUES  
548DO NOT SURVIVE THE OWNERSHIP CHANGE.” Question: If this is true, no delegations  
549work after a change of owner. How does the new owner set new AuthData values?

550 The simple way of handling this is to get AdminMonitor to own backing up delegations at  
551 first owner install and then be run by new owner, and AdminMonitor uses FillTable, to  
552 handle “Owner migration.” Or, for another use option, is for second owner to pick-up  
553 PCR-ID’s and delegations bits from previous owner – what is the most straight-forward  
554 way to do this?

555In section 3.1 (Delegate Definitions bit map table), several commands that do not require  
556owner authorization are in the table and can be delegated: TPM\_SetTempDeactivated (bit  
55715), TPM\_ReadPubek (bit 7), and TPM\_LoadManuMaintPub (bit 3), Why?

558In section 3.3 it is stated, “The Family ID resets to NULL on each change of TPM Owner.”  
559This invalidates all delegations. Is this what we want?

560 You don’t have to blow away FamilyID to blow away the blobs, because key is gone. So  
561 this is not required – can eliminate these actions.

562In section 3.12, why is TPM\_DELEGATE\_LABEL included in the table?

563In section 4.2 (TPM\_DelTable\_FillTable), is it okay to delete requirement that delegate table  
564be empty? Also, in Action 14, now that we have both persistent and volatile tableAdmin  
565flags, should this command set volatile tableAdmin flag to FALSE upon completion?

566 The delegate table does not need to be empty to use the TPM\_DelTable\_FillTable  
567 command, Also, a paragraph has been added to Informative comment for  
568 TPM\_DelTable\_FillTable that points out usefulness of immediately following  
569 TPM\_DelTable\_FillTable with TPM\_Delegate\_TempSetAdmin, to stop table administration  
570 in the current boot cycle.

571In section 4.15 (TPM\_FamTable\_IncrementCount), why does this command require  
572TPMOwner authorization, as currently documented in section 4.15?

573 IncrementCount is gated by tableAdmin, which seems sufficient, and use of ownerauth  
574 makes it difficult to automatically verify a table using a CDROM.

575In section 4.3 (TPM\_DelTable\_FillTableOwner), in the Action 3d, use OTP[80] = MFG(x1) in  
576place of oneTimePad[n] = SHA1(x1 || seed[n]))?,

577 yes.

578In section 4.9 (TPM\_DelTable\_SetRow), is invalidateRow input parameter really needed?

579 It is only used in action 5. Couldn't action 5 simply read "Set N1 -> familyID = NULL"?

580There is no easy way to generate a blob that can be used to delegate migration authority for  
581a user key.

582 This is because the TPM does not store the migration authority on the chip as the  
583 migration command involves an encrypted key, not a loaded one. One could invent a  
584 'CreateMigrationDelegationBlob' that took the encrypted key as input and generated the  
585 encrypted delegation blob as output, but it would not be pretty. Sorry Dave.

586If a delegate row in NV memory (nominally 4 rows) is to refer to a user key (instead of owner  
587auth), then it needs to include a hash of the public key. It could be that the NV table is  
588restricted to owner auth delegations, this would save 80 bytes of NV store and also simplify  
589the LoadBlob command.

590 Maybe would simplify other things. I would definitely NOT permit user keys in the table  
591 to be run with the legacy OSAP and OIAP ordinals.

592A few more GetCapability values are also required, the usual constants that we discussed  
593and also the two readTable caps.

594TBD Verify that Delegate Table Management commands (see section 2.8) cover all the  
595functionality of obsolete or updated commands.

596Redefine bits 16 and above in Delegation Definitions table (section 3.1). In particular, can  
597new command set (with TPM\_FAMILY\_OPERATION options as defined in section 3.20) be  
598delegated individually and appropriately. Also, how many user key authorized commands  
599will be delegated?

600Is new TPM\_FAMILY\_FLAGS field of family table (defined in section 3.5) sensitive data?

601DSAP informative comment needs to be completed (section 4.1). In particular, does the  
602statement "The DSAP command works like OSAP except it takes an encrypted blob – an  
603encrypted delegate table row -- as input" sufficient? Or do some particular differences  
604between DSAP and OSAP have to be pointed out in this informative comment??

605The TPM\_Delegate\_LoadBlob[Owner] commands cannot be used to load key delegation blobs  
606into the TPM. Is another ordinal required to do that?

607Is it okay for TPM\_Delegate\_LoadBlob[Owner] commands to ignore enable/disable  
608use/admin flags in family table rows?

609Is it wise to delegate TPM\_DelTable\_ConvertBlob command (defined in section 4.11)? Does  
610current definition of this command support section 2.7 scenarios?

611Is there a privacy problem with DelTable\_ReadRow since the contents may not be identical  
612from TPM to TPM?

613Are DSAP sessions being pooled with the other sessions? if so, can one save/load them by  
614context functions? if not, then there should be a restriction in saveContext.

615DSAP are "normal" authorization sessions and would save/load with OIAP and OSAP  
616sessions

617**End of informative comment**

101

## 6182.2.2 NV Questions

### 619Start of informative comment

620You would set this by using a new ordinal that is unauthorized and only turns the flag on to  
621lock everything. Yet another ordinal? Do we need it? Is this an important functionality for  
622the uses we see?

623 Yes this allows us to have "close" to write once functionality. What the functionality  
624 would be is that the RTM would assure that the proper information is present in the  
625 TPM and then "lock" the area. One could create this functionality by having the RTM  
626 change the authorization each time but then you would need to eat more NV store so  
627 save the sealed AuthData value. I think that is easier to have an ordinal than eat the NV  
628 space and require a much more complex programming model.

629Is it OK to have an element partially written?

630 Given that we have chunks there has to be a mechanism to allow partial writes.

631If an element is partially written, how does a caller know that more needs to be written?

632 I would say the use model that provides the ability to write – read, in a loop is just not  
633 supported. Get it all written and then do the read.

634Usage of the lock bit: as you wrote, the RTM would assure that the proper information is  
635present in the TPM and then "lock" the area. so why in action #4 we should also check  
636bWritten when the lock bit is set? should be as action #3b of TPM\_NV\_DefineSpace, if lock  
637is set - return error

638 [Grawrock, David] Not quite, the use model I was trying to create was the one where the  
639 TPM was locked and the user was attempting to add a new area. If the locked bit doesn't  
640 allow for writing once to a new area, one must reboot to perform the write and also tell  
641 the RTM what the value to write must be. So this allows the creator of an area to write it  
642 once and then it flows with the locked bit.

643Can you delete a NV value with only physical presence?

644 [Grawrock, David] You can't delete with physical presence, you must use owner  
645 authorization. This I think is a reasonable restriction to avoid burn problems.

646Why is there no check on the writes for a TPM Owner?

647 The check for an owner occurred during the TPM\_NV\_DefineSpace. It is imperative that  
648 the TPM\_NV\_DefineSpace set in place the appropriate restrictions to limit the potential  
649 for attacks on the NV storage area.

650Description of maxNVBufSize is confusing to me. Why is this value related to the input size?  
651And since there is no longer any 'written' bits, why is there a maximum area size at all?

652 [Grawrock, David] This is a fixed size and set by the TPM manufacturer. I would see  
653 values like the input buffer, transport sessions etc all coming up with the max size the  
654 TPM can handle. This does NOT indicate what is available on the TPM right now. The  
655 TPM could have 4k of space but max size would be 782 and would always report that  
656 number. If the available space fell to 20 bytes this value would still be 782.

657If the storage area is an opaque area to the TPM (as described), then how does the TPM  
658know what PCR registers have been used to seal a blob?

659 The VALUES of the area are opaque, the attributes to control access are not. So if the  
660 attributes indicate that PCR restrictions are in place the TPM keeps those PCR values as  
661 part of the index attributes. This in reality seals the value as there is no need for  
662 tpmProof since the value never leaves the TPM.

663**End of informative comment**



## 664**3. Protection**

### 665**3.1 Introduction**

#### 666**Start of informative comment**

667The Protection Profile in the Conformance part of the specification defines the threats that  
668are resisted by a platform. This section, “Protection,” describes the properties of selected  
669capabilities and selected data locations within a TPM that has a Protection Profile and has  
670not been modified by physical means.

671This section introduces the concept of protected capabilities and the concept of shielded  
672locations for data. The ordinal set defined in part II and III is the set of protected  
673capabilities. The data structures in part II define the shielded locations.

674• A protected capability is one whose correct operation is necessary in order for the  
675operation of the TCG Subsystem to be trusted.

676• A shielded location is an area where data is protected against interference and prying,  
677independent of its form.

678This specification uses the concept of protected capabilities so as to distinguish platform  
679capabilities that must be trustworthy. Trust in the TPM depends critically on the protected  
680capabilities. Platform capabilities that are not protected capabilities must (of course) work  
681properly if the TCG Subsystem is to function properly.

682This specification uses the concept of shielded locations, rather than the concept of  
683“shielded data.” While the concept of shielded data is intuitive, it is extraordinarily difficult  
684to define because of the imprecise meaning of the word “data.” For example, consider data  
685that is produced in a safe location and then moved into ordinary storage. It is the same data  
686in both locations, but in one it is shielded data and in the other it is not. Also, data may not  
687always exist in the same form. For example, it may exist as vulnerable plaintext, but also  
688may sometimes be transformed into a logically protected form. This data continues to exist,  
689but doesn't always need to be shielded data - the vulnerable form needs to be shielded data,  
690but the logically protected form does not. If a specific form of data requires protection  
691against interference or prying, it is therefore necessary to say “if the data-D exists, it must  
692exist only in a shielded location.” A more concise expression is “the data-D must be extant  
693only in a shielded location.”

694Hence, if trust in the TCG Subsystem depends critically on access to certain data, that data  
695should be extant only in a shielded location and accessible only to protected capabilities.  
696When not in use, such data could be erased after conversion (using a protected capability)  
697into another data structure. Unless the other data structure was defined as one that must  
698be held in a shielded location, it need not be held in a shielded location.

#### 699**End of informative comment**

7001. The data structures described in part II of the TPM specifications MUST NOT be  
701 instantiated in a TPM, except as data in TPM-shielded-locations.

7022. The ordinal set defined in part II and III of the TPM specifications MUST NOT be  
703 instantiated in a TPM, except as TPM-protected-capabilities.

7043. Functions MUST NOT be instantiated in a TPM as TPM-protected-capabilities if they do  
705 not appear in the ordinal set defined in part II and III of the TPM specifications.

### 706**3.2 Threat**

#### 707**Start of informative comment**

708This section, “Threat,” defines the scope of the threats that must be considered when  
709considering whether a platform facilitates subversion of capabilities and data in a platform.

710The design and implementation of a platform determines the extent to which the platform  
711facilitates subversion of capabilities and data within that platform. It is necessary to define  
712the attacks that must be resisted by TPM-shielded locations and TPM-protected capabilities  
713in that platform.

714The TCG specifications define all attacks that are resisted by the TPM. These attacks must  
715be considered when determining whether the integrity of TPM-protected capabilities and  
716data in TPM-shielded locations can be damaged. These attacks must be considered when  
717determining whether there is a backdoor method of obtaining access to TPM-protected  
718capabilities and data in TPM-shielded locations. These attacks must be considered when  
719determining whether TPM-protected capabilities have undesirable side effects.

#### 720**End of informative comment**

7211. For the purposes of the “Protection” section of the specification, the threats that MUST  
722 be considered when determining whether the TPM facilitates subversion of TPM-  
723 protected-capabilities or data in TPM-shielded-locations SHALL include

724 a. The methods inherent in physical attacks that fail if the TPM complies with the  
725 “physical protection” requirements specified by TCG

726 b. All methods that require execution of instructions in a computing engine in the  
727 platform

### 728**3.3 Protection of functions**

#### 729**Start of informative comment**

730A TPM-protected-capability must be used to modify TPM-protected capabilities. Other  
731methods must not be allowed to modify TPM-protected capabilities. Otherwise, the integrity  
732of TPM-protected capabilities is unknown.

#### 733**End of informative comment**

7341. A TPM SHALL NOT facilitate the alteration of TPM-protected-capabilities, except by TPM-  
735 protected capabilities.

### 736**3.4 Protection of information**

#### 737**Start of informative comment**

738TPM-protected capabilities must provide the only means from outside the TPM to access  
739information represented by data in TPM-shielded-locations. Otherwise, a rogue can reveal  
740data in TPM-shielded-locations, or create a derivative of data from TPM-shielded-locations  
741(in a way that maintains some or all of the information content of the data) and reveal the  
742derivative.

**743End of informative comment**

7441. A TPM SHALL NOT export data that is dependent upon data structures described in part  
745 II of the TPM specifications, other than via a TPM-Protected-Capability.

**7463.5 Side effects****747Start of informative comment**

748An implementation of a TPM-protected capability must not disclose the contents of TPM-  
749shielded locations. The only exceptions are when such disclosure is inherent in the  
750definition of the capability or in the methods used by the capability. For example, a  
751capability might be designed specifically to reveal hidden data or might use cryptography  
752and hence always be vulnerable to cryptanalysis. In such cases, some disclosure or risk of  
753disclosure is inherent and cannot be avoided. Other forms of disclosure (by side effects, for  
754example) must always be avoided.

**755End of informative comment**

7561. The implementation of a TPM-protected-capability in a TPM SHALL NOT facilitate the  
757 disclosure or the exposure of information represented by data in TPM-shielded-  
758 locations, except by means unavoidably inherent in the TPM definition.

**7593.6 Exceptions and clarifications****760Start of informative comment**

761These exceptions to the blanket statements in the generic “protection” requirements (above)  
762are fully compatible with the intended effect of those statements. These exceptions affect  
763TCG-data that is available as plain-text outside the TPM and TCG-data that can be used  
764without violating security or privacy. These exceptions are valuable because they approve  
765use of TPM resources by vendor-specific commands in particular circumstances.

766These clarifications to the blanket statements of the generic “protection” requirements  
767(above) do not materially change the effect of those statements, but serve to approve specific  
768legitimate interpretations of the requirements.

**769End of informative comment**

7701. A Shielded Location is a place (memory, register, etc.) where data is protected against  
771 interference and exposure, independent of its form

7722. A TPM-Protected-Capability is an operation defined in and restricted to those identified  
773 in part II and III of the TPM specifications.

7743. A vendor specific command or capability MAY use the standard TCG owner/operator  
775 authorization mechanism

7764. A vendor specific command or capability MAY utilize a TPM\_PUBKEY structure stored on  
777 the TPM so long as the usage of that TPM\_PUBKEY structure is authorized using the  
778 standard TCG authorization mechanism.

7795. A vendor specific command or capability MAY use a sequence of standard TCG  
780 commands. The command MUST propagate the locality used for the call to the used  
781 TCG commands or capabilities, or set locality to 0.

7826. A vendor specific command or capability that takes advantage of exceptions and  
783 clarifications to the “protection” requirements MUST be defined as part of the security  
784 target of the TPM. Such a vendor specific command or capability MUST be evaluated to  
785 meet the Platform Specific TPM and System Security Targets.
7867. If a TPM employs vendor-specific cipher-text that is protected against subversion to the  
787 same or greater extent as internal TPM-resources stored outside the TPM with TCG-  
788 defined methods, that vendor-specific cipher-text does not necessarily require protection  
789 from physical attack. If a TPM location stores only vendor-specific cipher-text that does  
790 not require protection from physical attack, that location can be ignored when  
791 determining whether the TPM complies with the "physical protection" requirements  
792 specified by TCG.

## 7934. TPM Architecture

### 794 4.1 Interoperability

#### 795Start of informative comment

796The TPM must support a minimum set of algorithms and operations to meet TCG  
797specifications.

798Algorithms

799RSA, SHA-1, HMAC

800The algorithms and protocols are the minimum that the TPM must support. Additional  
801algorithms and protocols may be available to the TPM. All algorithms and protocols  
802available in the TPM must be included in the TPM and platform credential.

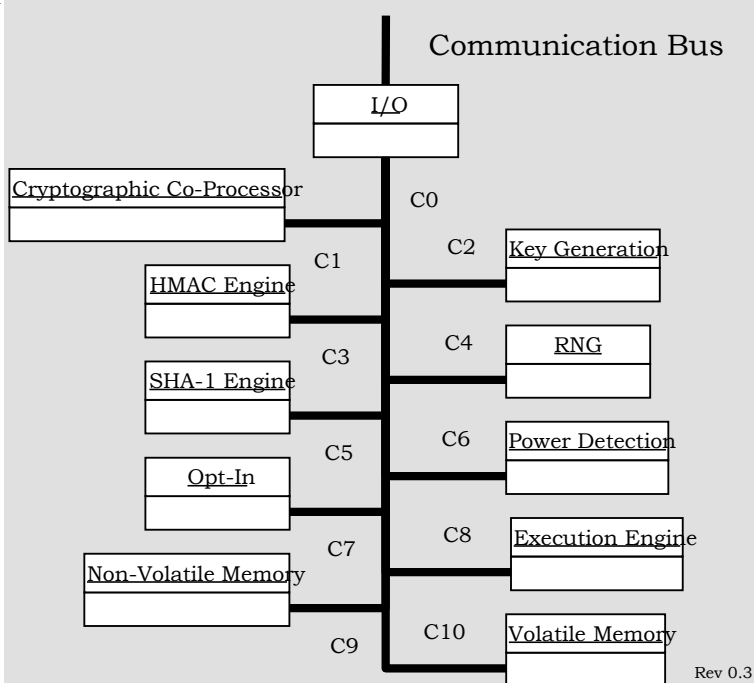
803The reason to specify these algorithms is two fold. The first is to know and understand the  
804security properties of selected algorithms; identify appropriate key sizes and ensure  
805appropriate use in protocols. The second reason is to define a base level of algorithms for  
806interoperability.

#### 807End of informative comment

### 808 4.2 Components

#### 809Start of informative comment

810The following is a block diagram Figure 4:a shows the major components of a TPM.  
811



812Figure 4:a - TPM Component Architecture

813 **End of informative comment**

814 **4.2.1 Input and Output**

815 **Start of informative comment**

816 The I/O component, Figure 4:a C0, manages information flow over the communications  
817 bus. It performs protocol encoding/decoding suitable for communication over external and  
818 internal buses. It routes messages to appropriate components. The I/O component enforces  
819 access policies associated with the Opt-In component as well as other TPM functions  
820 requiring access control.

821 The main specification does not require a specific I/O bus. Issues around a particular I/O  
822 bus are the purview of a platform specific specification.

823 **End of informative comment**

- 824 1. The number of incoming operand parameter bytes must exactly match the  
825 requirements of the command ordinal. If the command contains more or fewer bytes  
826 than required, the TPM MUST return TPM\_BAD\_PARAMETER.

827 **4.2.2 Cryptographic Co-Processor**

828 **Start of informative comment**

829 The cryptographic co-processor, Figure 4:a C1, implements cryptographic operations within  
830 the TPM. The TPM employs conventional cryptographic operations in conventional ways.  
831 Those operations include the following:

832 Asymmetric key generation (RSA)

833 Asymmetric encryption/decryption (RSA)

834 Hashing (SHA-1)

835 Random number generation (RNG)

836 The TPM uses these capabilities to perform generation of random data, generation of  
837 asymmetric keys, signing and confidentiality of stored data.

838 The TPM may symmetric encryption for internal TPM use but does not expose any  
839 symmetric algorithm functions to general users of the TPM.

840 The TPM may implement additional asymmetric algorithms. TPM devices that implement  
841 different algorithms may have different algorithms perform the signing and wrapping.

842 If the TPM uses RSA with the required key length (2048 bits for storage keys), the output of  
843 all commands for key or data blob generation (e.g., TPM\_CreateWrapKey, TPM\_Seal,  
844 TPM\_Sealx, TPM\_MakeIdentity) consists of only one block. However, if the TPM uses other  
845 asymmetric algorithms that result in more than one output block for these commands, the  
846 integrity of the blobs must be protected by the TPM (by means of appropriate chaining  
847 mechanisms).

848 **End of informative comment**

- 849 1. The TPM MAY implement other asymmetric algorithms such as DSA or elliptic curve.

- 137
- 850 a. These algorithms may be in use for wrapping, signatures and other operations. There  
851 is no guarantee that these keys can migrate to other TPM devices or that other TPM  
852 devices will accept signatures from these additional algorithms.
- 853 b. If the output key or data blob generated with a storage key consists of more than one  
854 block, the TPM MUST protect the integrity of the blob by means of appropriate  
855 chaining mechanisms.
8562. All storage keys MUST be of strength equivalent to a 2048 bits RSA key or greater. The  
857 TPM SHALL NOT load a storage key whose strength less than that of a 2048 bits RSA  
858 key.
8593. All AIK MUST be of strength equivalent to a 2048 bits RSA key, or greater.

#### 860 4.2.2.1 RSA Engine

##### 861 **Start of informative comment**

862The RSA asymmetric algorithm is used for digital signatures and for encryption.  
863For RSA keys the PKCS #1 standard provides the implementation details for digital  
864signature, encryption and data formats.

865There is no requirement concerning how the RSA algorithm is to be implemented. TPM  
866manufacturers may use Chinese Remainder Theorem (CRT) implementations or any other  
867method. Designers should review P1363 for guidance on RSA implementations.

868For keys that are required to be 2048-bit RSA keys, the default  $2^{16}+1$  exponent will be  
869required. This guarantees the strength of the key without walking a hierarchy that cannot  
870necessarily be walked reliably.

##### 871 **End of informative comment**

8721. The TPM MUST support RSA.
8732. The TPM MUST use the RSA algorithm for encryption and digital signatures.
8743. The TPM MUST support key sizes of 512, 1024, and 2048 bits. The TPM MAY support  
875 other key sizes.
- 876 a. The minimum RECOMMENDED key size is 2048 bits.
- 877 b. In FIPS mode, the minimum key size MUST be 1024.
8784. The TPM MUST support an RSA public exponent of  $2^{16}+1$ . The TPM MAY support other  
879 exponent values.
8805. TPM devices that use CRT as the RSA implementation MUST provide protection and  
881 detection of failures during the CRT process to avoid attacks on the private key.

#### 882 4.2.2.2 Signature Operations

##### 883 **Start of informative comment**

884The TPM performs signatures on both internal items and on requested external blobs. The  
885rules for signatures apply to both operations.

##### 886 **End of informative comment**

8871. The TPM MUST use the RSA algorithm for signature operations where signed data is  
888 verified by entities other than the TPM that performed the sign operation.

8892. The TPM MAY use other asymmetric algorithms for signatures; however, there is no  
890 requirement that other TPM devices either accept or verify those signatures.

8913. The TPM MUST use P1363 for the format and design of the signature output.

### 892 4.2.2.3 Symmetric Encryption Engine

#### 893 **Start of informative comment**

894The TPM uses symmetric encryption to encrypt authentication information, provide  
895confidentiality in transport sessions and provide internal encryption of blobs stored off the  
896TPM.

897For authentication and transport sessions, the mandatory mechanism is a Vernam one-  
898time-pad with XOR. The mechanism to generate the one-time-pad is MGF1 and the nonces  
899from the session protocol. When encrypting authorization data, the authorization data and  
900the nonces are the same size, 20 bytes, so a direct XOR is possible.

901For transport sessions the size of data is larger than the nonces so there needs to be a  
902mechanism to expand the entropy to the size of the data. The mechanism to expand the  
903entropy is the MGF1 function from PKCS#1. This function provides a known mechanism  
904that does not lower the entropy of the nonces.

905AES may be supported as an alternate symmetric key encryption algorithm.

906Internal protection of information can use any symmetric algorithm that the TPM designer  
907feels provides the proper level of protection.

908The TPM does not expose any of the symmetric operations for general message encryption.

#### 909 **End of informative comment**

### 910 4.2.2.4 Using Keys

#### 911 **Start of Informative comments:**

912Keys can be symmetric or asymmetric.

913As the TPM does not have an exposed symmetric algorithm, the TPM is only a generator,  
914storage device and protector of symmetric keys. Generation of the symmetric key would use  
915the TPM RNG. Storage and protection would be provided by the BIND and SEAL capabilities  
916of the TPM. If the caller wants to ensure that the release of a symmetric key is not exposed  
917after UNBIND/UNSEAL on delivery to the caller, the caller should use a transport session  
918with confidentiality set.

919For asymmetric algorithms, the TPM generates and operates on RSA keys. The keys can be  
920held only by the TPM or in conjunction with the caller of the TPM. If the private portion of a  
921key is in use outside of the TPM it is the responsibility of the caller and user of that key to  
922ensure the protections of the key.

923The TPM has provisions to indicate if a key is held exclusively for the TPM or can be shared  
924with entities off of the TPM.

#### 925 **End of informative comments.**



- 146
9261. A secret key is a key that is a private asymmetric key or a symmetric key.
9272. Data SHOULD NOT be used as a secret key by a TCG protected capability unless that  
928 data has been extant only in a shielded location.
9293. A key generated by a TCG protected capability SHALL NOT be used as a secret key  
930 unless that key has been extant only in a shielded location.
9314. A secret key obtained by a TCG protected capability from a Protected Storage blob  
932 SHALL be extant only in a shielded location.

### 933 4.2.3 Key Generation

#### 934Start of informative comment

935The Key Generation component, Figure 4:a C2, creates RSA key pairs and symmetric keys.  
936TCG places no minimum requirements on key generation times for asymmetric or  
937symmetric keys.

#### 938End of informative comment

### 939 4.2.3.1 Asymmetric – RSA

940The TPM MUST generate asymmetric key pairs. The generate function is a protected  
941capability and the private key is held in a shielded location. The implementation of the  
942generate function MUST be in accordance with P1363.

943The prime-number testing for the RSA algorithm MUST use the definitions of P1363. If  
944additional asymmetric algorithms are available, they MUST use the definitions from P1363  
945for the underlying basis of the asymmetric key (for example, elliptic curve fitting).

### 946 4.2.3.2 Nonce Creation

947The creation of all nonce values MUST use the next n bits from the TPM RNG.

### 948 4.2.4 HMAC Engine

#### 949Start of informative comment

950The HMAC engine, Figure 4:a C3, provides two pieces of information to the TPM: proof of  
951knowledge of the AuthData and proof that the request arriving is authorized and has no  
952modifications made to the command in transit.

953The HMAC definition is for the HMAC calculation only. It does not specify the order or  
954mechanism that transports the data from caller to actual TPM.

955The creation of the HMAC is order dependent. Each command has specific items that are  
956portions of the HMAC calculation. The actual calculation starts with the definition from  
957RFC 2104.

958RFC 2104 requires the selection of two parameters to properly define the HMAC in use.  
959These values are the key length and the block size. This specification will use a key length  
960of 20 bytes and a block size of 64 bytes. These values are known in the RFC as K for the key  
961length and B as the block size.

962The basic construct is

963  $H(K \text{ XOR opad}, H(K \text{ XOR ipad}, \text{text}))$

964 where

965  $H$  = the SHA1 hash operation

966  $K$  = the key or the AuthData

967  $\text{XOR}$  = the xor operation

968  $\text{opad}$  = the byte 0x5C repeated  $B$  times

969  $B$  = the block length

970  $\text{ipad}$  = the byte 0x36 repeated  $B$  times

971  $\text{text}$  = the message information and any parameters from the command

972 **End of informative comment**

973 The TPM MUST support the calculation of an HMAC according to RFC 2104.

974 The size of the key ( $K$  in RFC 2104) MUST be 20 bytes. The block size ( $B$  in RFC 2104)  
975 MUST be 64 bytes.

976 The order of the parameters is critical to the TPM's ability to recreate the HMAC. Not all of  
977 the fields are sent on the wire for each command for instance only one of the nonce values  
978 travels on the wire. Each command interface definition indicates what parameters are  
979 involved in the HMAC calculation.

980 **4.2.5 Random Number Generator**

981 **Start of informative comment**

982 The Random Number Generator (RNG) component, Figure 6:a C4 is the source of  
983 randomness in the TPM. The TPM uses these random values for nonces, key generation,  
984 and randomness in signatures.

985 The RNG consists of a state-machine that accepts and mixes unpredictable data and a post-  
986 processor that has a one-way function (e.g. SHA-1). The idea behind the design is that a  
987 TPM can be good source of randomness without having to require a genuine source of  
988 hardware entropy.

989 The state-machine can have a non-volatile state initialized with unpredictable random data  
990 during TPM manufacturing before delivery of the TPM to the customers. The state-machine  
991 can accept, at any time, further (unpredictable) data, or entropy, to salt the random  
992 number. Such data comes from hardware or software sources – for example; from thermal  
993 noise, or by monitoring random keyboard strokes or mouse movements. The RNG requires a  
994 reseeding after each reset of the TPM. A true hardware source of entropy is likely to supply  
995 entropy at a higher baud rate than a software source.

996 When adding entropy to the state-machine, the process must ensure that after the addition,  
997 no outside source can gain any visibility into the new state of the state-machine. Neither  
998 the Owner of the TPM nor the manufacturer of the TPM can deduce the state of the state-  
999 machine after shipment of the TPM. The RNG post-processor condenses the output of the  
1000 state-machine into data that has sufficient and uniform entropy. The one-way function  
1001 should use more bits of input data than it produces as output.

155

1002Our definition of the RNG allows implementation of a Pseudo Random Number Generator  
1003(PRNG) algorithm. However, on devices where a hardware source of entropy is available, a  
1004PRNG need not be implemented. This specification refers to both RNG and PRNG  
1005implementations as the RNG mechanism. There is no need to distinguish between the two  
1006at the TCG specification level.

1007The TPM should be able to provide 32 bytes of randomness on each call. Larger requests  
1008may fail with not enough randomness being available.

1009**End of informative comment**

10101. The RNG for the TPM will consist of the following components:

- 1011 a. Entropy source and collector
- 1012 b. State register
- 1013 c. Mixing function

10142. The RNG capability is a TPM-protected capability with no access control.

10153. The RNG output may or may not be shielded data. When the data is for internal use by  
1016 the TPM (e.g., generation of tpmProof or an asymmetric key), the data **MUST** be held in a  
1017 shielded location. The RNG output for internal use **MUST** not be known outside the  
1018 TPM. In particular, it **MUST** not be known by the TPM manufacturer. When the data is  
1019 for use by the TSS or another external caller, the data is not shielded.

10204. In FIPS mode, the RNG **MUST** be a NIST approved RNG. The NIST self-test requirements  
1021 **MUST** be satisfied.

1022 **4.2.5.1 Entropy Source and Collector**

1023**Start of informative comment**

1024The entropy source is the process or processes that provide entropy. These types of sources  
1025could include noise, clock variations, air movement, and other types of events.

1026The entropy collector is the process that collects the entropy, removes bias, and smoothes  
1027the output. The collector differs from the mixing function in that the collector may have  
1028special code to handle any bias or skewing of the raw entropy data. For instance, if the  
1029entropy source has a bias of creating 60 percent 1s and only 40 percent 0s, then the  
1030collector design takes that bias into account before sending the information to the state  
1031register.

1032**End of informative comment**

10331. The entropy source **MUST** provide entropy to the state register in a manner that provides  
1034 entropy that is not visible to an outside process.

- 1035 a. For compliance purposes, the entropy source **MAY** be outside of the TPM; however,  
1036 attention **MUST** be paid to the reporting mechanism.

10372. The entropy source **MUST** provide the information only to the state register.

- 1038 a. The entropy source may provide information that has a bias, so the entropy collector  
1039 must remove the bias before updating the state register. The bias removal could use  
1040 the mixing function or a function specifically designed to handle the bias of the  
1041 entropy source.

- 1042 b. The entropy source can be a single device (such as hardware noise) or a combination  
1043 of events (such as disk timings). It is the responsibility of the entropy collector to  
1044 update the state register whenever the collector has additional entropy.

#### 1045 4.2.5.2 State Register

##### 1046 **Start of informative comment**

1047 The state register implementation may use two registers: a non-volatile register rngState  
1048 and a volatile register. The TPM loads the volatile register from the non-volatile register on  
1049 startup. Each subsequent change to the state register from either the entropy source or the  
1050 mixing function affects the volatile state register. The TPM saves the current value of the  
1051 volatile state register to the non-volatile register on TPM power-down. The TPM may update  
1052 the non-volatile register at any other time. The reasons for using two registers are:

- 1053 To handle an implementation in which the non-volatile register is in a flash device;  
1054 To avoid overuse of the flash, as the number of writes to a flash device are limited.

##### 1055 **End of informative comment**

1056 1. The state register is in a TPM shielded-location.

- 1057 a. The state register MUST be non-volatile.  
1058 b. The update function to the state register is a TPM protected-capability.  
1059 c. The primary input to the update function SHOULD be the entropy collector.  
1060 2. If the current value of the state register is unknown, calls made to the update function  
1061 with known data MUST NOT result in the state register ending up in a state that an  
1062 attacker could know.  
1063 a. This requirement implies that the addition of known data MUST NOT result in a  
1064 decrease in the entropy of the state register.  
1065 3. The TPM MUST NOT export the state register.

#### 1066 4.2.5.3 Mixing Function

##### 1067 **Start of informative comment**

1068 The mixing function takes the state register and produces output. The mixing function is a  
1069 TPM protected-capability. The mixing function takes the value from a state register and  
1070 creates the RNG output. If the entropy source has a bias, then the collector takes that bias  
1071 into account before sending the information to the state register.

##### 1072 **End of informative comment**

1073 1. Each use of the mixing function MUST affect the state register.

- 1074 a. This requirement is to affect the volatile register and does not need to affect the non-  
1075 volatile state register.

#### 1076 4.2.5.4 RNG Reset

##### 1077 **Start of informative comment**

1078 The resetting of the RNG occurs at least in response to a loss of power to the device.

1079These tests prove only that the RNG is still operating properly; they do not prove how much  
 1080entropy is in the state register. This is why the self-test checks only after the load of  
 1081previous state and may occur before the addition of more entropy.

#### 1082**End of informative comment**

10831. The RNG MUST NOT output any bits after a system reset until the following occurs:

- 1084 a. The entropy collector performs an update on the state register. This does not include  
 1085 the adding of the previous state but requires at least one bit of entropy.
- 1086 b. The mixing function performs a self-test. This self-test MUST occur after the loading  
 1087 of the previous state. It MAY occur before the entropy collector performs the first  
 1088 update.

### 1089 **4.2.6 SHA-1 Engine**

#### 1090**Start of informative comment**

1091The SHA-1, Figure 4:a C5, hash capability is primarily used by the TPM, as it is a trusted  
 1092implementation of a hash algorithm. The hash interfaces are exposed outside the TPM to  
 1093support Measurement taking during platform boot phases and to allow environments that  
 1094have limited capabilities access to a hash functions. The TPM is not a cryptographic  
 1095accelerator. TCG does not specify minimum throughput requirements for TPM hash  
 1096services.

#### 1097**End of informative comment**

10981. The TPM MUST implement the SHA-1 hash algorithm as defined by FIPS-180-1.

10992. The output of SHA-1 is 160 bits and all areas that expect a hash value are REQUIRED  
 1100 to support the full 160 bits.

11013. The only commands that SHALL be presented to the TPM in-between a TPM\_SHA1Start  
 1102 command and a TPM\_SHA1Complete command SHALL be a variable number (possibly  
 1103 0) of TPM\_SHA1Update commands.

1104 a. The TPM\_SHA1Update commands can occur in a transport session.

11054. Throughout all parts of the specification the characters x1 || x2 imply the  
 1106 concatenation of x1 and x2

### 1107 **4.2.7 Power Detection**

#### 1108**Start of informative comment**

1109The power detection component, Figure 4:a C6, manages the TPM power states in  
 1110conjunction with platform power states. TCG requires that the TPM be notified of all power  
 1111state changes.

1112Power detection also supports physical presence assertions. The TPM may restrict  
 1113command-execution during periods when the operation of the platform is physically  
 1114constrained. In a PC, operational constraints occur during the power-on self-test (POST)  
 1115and require Operator input via the keyboard. The TPM might allow access to certain  
 1116commands while in a constrained execution mode or boot state. At some critical point in the  
 1117POST process, the TPM may be notified of state changes that affect TPM command  
 1118processing modes.

1119**End of informative comment**

1120 **4.2.8 Opt-In**

1121**Start of informative comment**

1122The Opt-In component, Figure 4:a C7, provides mechanisms and protections to allow the  
1123TPM to be turned on/off, enabled/disabled, activated/deactivated. The Opt-In component  
1124maintains the state of persistent and volatile flags and enforces the semantics associated  
1125with these flags.

1126The setting of flags requires either authorization by the TPM Owner or the assertion of  
1127physical presence at the platform. The platform's manufacturer determines the techniques  
1128used to represent physical-presence. The guiding principle is that no remote entity should  
1129be able to change TPM status without either knowledge of the TPM Owner or the Operator is  
1130physically present at the platform. Physical presence may be asserted during a period when  
1131platform operation is constrained such as power-up.

1132Non-Volatile Flags:

1133physicalPresenceLifetimeLock

1134physicalPresenceHWEnable

1135physicalPresenceCMDEnable

1136Volatile Flags:

1137physicalPresenceLock

1138physicalPresence

1139The notation physicalPresenceV indicates the physical presence state that ordinals refer to  
1140when they say, for example, "if physical presence is asserted".

173

1141

1142The following truth table explains the conditions in which the physicalPresenceV flag may  
1143be altered:

Persistent / Volatile	P	P	P	V	
Control Flags	physicalPresenceLifetimeLock	physicalPresenceHWEEnable	physicalPresenceCMDEnable	physicalPresenceLock	
Volatile Access Semantics to Physical Presence Flag	-	F	F	-	No access to physicalPresenceV flag.
	-	F	-	T	
	-	-	T	F	Access to physicalPresenceV flag through TCS_PhysicalPresence command enabled.
	-	T	-	-	Access to physicalPresenceV flag through hardware signal enabled.
	-	T	T	F	Access to physicalPresenceV flag through hardware signal or TCS_PhysicalPresence command enabled.
Persistent Access Semantics to Physical Presence Flag	T	F	F	-	Access to physicalPresenceV flag permanently disabled.
	T	F	T	T	Access to physicalPresenceV flag disabled until TPM_Startup(ST_CLEAR).
	T	F	T	F	Exclusive access to physicalPresenceV flag through TCS_PhysicalPresence command permanently enabled.
	T	T	F	-	Exclusive access to physicalPresenceV flag through hardware signal permanently enabled.
	T	T	T	F	Access to physicalPresenceV flag through hardware signal or TCS_PhysicalPresence command permanently enabled.

1144

1145Table 4:a - Physical Presence Semantics

1146TCG also recognizes the concept of unambiguous physical presence. Conceptually, the use  
1147of dedicated electrical hardware providing a trusted path to the Operator has higher  
1148precedence than the physicalPresenceV flag value. Unambiguous physical presence may be  
1149used to override physicalPresenceV flag value under conditions specified by platform  
1150specific design considerations.

1151Additional details relating to physical presence can be found in sections on Volatile and  
1152Non-volatile memory.

1153**End of informative comment**

## 1154 4.2.9 Execution Engine

1155**Start of informative comment**

1156The execution engine, Figure 4:a C8, runs program code to execute the TPM commands  
1157received from the I/O port. The execution engine is a vital component in ensuring that  
1158operations are properly segregated and shield locations are protected.

1159 **End of informative comment**

1160 **4.2.10 Non-Volatile Memory**

1161 **Start of informative comment**

1162 Non-volatile memory component, Figure 4:a C9, is used to store persistent identity and  
1163 state associated with the TPM. The NV area has set items (like the EK) and also is available  
1164 for allocation and use by entities authorized by the TPM Owner.

1165 The TPM designer should consider the use model of the TPM and if the use of NV storage is  
1166 a concern. NV storage does have a limited life and using the NV storage in a high volume  
1167 use model may prematurely wear out the TPM.

1168 There is no requirement for the TPM to protect against wear out by detecting that a write of  
1169 the same value need not be performed. Applications should avoid frequent writes of the  
1170 same value. For example, precede a TPM\_SetCapability with a TPM\_GetCapability and skip  
1171 the write if the TPM already holds the desired value.

1172 **End of informative comment**

1173 **4.3 Data Integrity Register (DIR)**

1174 **Start of informative comment**

1175 The DIR were a version 1.1 function. They provided a place to store information using the  
1176 TPM NV storage.

1177 In 1.2 the DIR are deprecated and the use of the DIR should move to the general purpose  
1178 NV storage area.

1179 The TPM must still support the functionality of the DIR register in the NV storage area.

1180 **End of informative comment**

1181 1. A TPM MUST provide one Data Integrity Register (DIR)

1182 a. The TPM DIR commands are deprecated in 1.2

1183 b. The TPM MUST reserve the space for one DIR in the NV storage area

1184 c. The TPM MAY have more than 1 DIR.

1185 2. The DIR MUST be 160-bit values and MUST be held in TPM shielded-locations.

1186 3. The DIR MUST be non-volatile (values are maintained during the power-off state).

1187 a. A TPM implementation need not provide the same number of DIRs as PCRs.

1188 **4.4 Platform Configuration Register (PCR)**

1189 **Start of informative comment**

1190 A Platform Configuration Register (PCR) is a 160-bit storage location for discrete integrity  
1191 measurements. There are a minimum of 16 PCR registers. All PCR registers are shielded-  
1192 locations and are inside of the TPM. The decision of whether a PCR contains a standard  
1193 measurement or if the PCR is available for general use is deferred to the platform specific  
1194 specification.



1195A large number of integrity metrics may be measured in a platform, and a particular  
 1196integrity metric may change with time and a new value may need to be stored. It is difficult  
 1197to authenticate the source of measurement of integrity metrics, and as a result a new value  
 1198of an integrity metric cannot be permitted to simply overwrite an existing value. (A rogue  
 1199could erase an existing value that indicates subversion and replace it with a benign value.)  
 1200Thus, if values of integrity metrics are individually stored, and updates of integrity metrics  
 1201must be individually stored, it is difficult to place an upper bound on the size of memory  
 1202that is required to store integrity metrics.

1203The PCR is designed to hold an unlimited number of measurements in the register. It does  
 1204this by using a cryptographic hash and hashing all updates to a PCR. The pseudo code for  
 1205this is:

1206     PCR<sub>i</sub> New = HASH ( PCR<sub>i</sub> Old value || value to add)

1207There are two salient properties of cryptographic hash that relate to PCR construction.  
 1208Ordering – meaning updates to PCRs are not commutative. For example, measuring (A then  
 1209B) is not the same as measuring (B then A).

1210The other hash property is one-way-ness. This property means it should be computationally  
 1211infeasible for an attacker to determine the input message given a PCR value. Furthermore,  
 1212subsequent updates to a PCR cannot be determined without knowledge of the previous PCR  
 1213values or all previous input messages provided to a PCR register since the last reset.

1214If the TPM is disabled or deactivated, commands that extend a PCR (e.g., TPM\_Extend,  
 1215TPM\_SHA1CompleteExtend) return a PCR value of all zeros, and commands that use the  
 1216PCRs (e.g., TPM\_PCRRead) are not available. However, the commands that extend a PCR  
 1217still update the PCR correctly and return success. For disabled, this is because the TPM  
 1218may become enabled later, and so must not miss a measurement. For deactivated, this is  
 1219because resource limited code like the CRTM will perform extends and may not be able to  
 1220handle a deactivated error case.

#### 1221 **End of informative comment**

12221. The PCR MUST be a 160-bit field that holds a cumulatively updated hash value

12232. The PCR MUST have a status field associated with it

12243. The PCR MUST be in the RTS and should be in volatile storage

12254. The PCR MUST allow for an unlimited number of measurements to be stored in the PCR

12265. The PCR MUST preserve the ordering of measurements presented to it

12276. A PCR MUST be set to the default value as specified by the PCRReset attribute

12287. A TPM implementation MUST provide 16 or more independent PCRs. These PCRs are  
 1229 identified by index and MUST be numbered from 0 (that is, PCR0 through PCR15 are  
 1230 required for TCG compliance). Vendors MAY implement more registers for general-  
 1231 purpose use. Extra registers MUST be numbered contiguously from 16 up to max – 1,  
 1232 where max is the maximum offered by the TPM.

12338. The TCG-protected capabilities that expose and modify the PCRs use a 32-bit index,  
 1234 indicating the maximum usable PCR index. However, TCG reserves register indices 230  
 1235 and higher for later versions of the specification. A TPM implementation MUST NOT  
 1236 provide registers with indices greater than or equal to 230. In this specification, the  
 1237 following terminology is used (although this internal format is not mandated).

12389. The PSS MUST define at least one measurement that the RTM MUST make and  
1239 the PCR where the measurement is stored.

124010. A TCG measurement agent MAY discard a duplicate event instead of incorporating it in a  
1241 PCR, provided that:

124211. A relevant TCG platform specification explicitly permits duplicates of this type of event to  
1243 be discarded

124412. The PCR already incorporates at least one event of this type

124513. An event of this type previously incorporated into the PCR included a statement that  
1246 duplicate such events may be discarded. This option could be used where frequent  
1247 recording of sleep states will adversely affect the lifetime of a TPM, for example.

124814. PCRs and the protected capabilities that operate upon them MAY NOT be used until  
1249 power-on self-test (TPM POST) has completed. If TPM POST fails, the TPM\_Extend  
1250 operation will fail; and, of greater importance, the TPM\_Quote operation and TPM\_Seal  
1251 operations that respectively report and examine the PCR contents MUST fail. At the  
1252 successful completion of TPM POST, all PCRs MUST be set to their default value (either  
1253 0x00...00 or 0xFF...FF). Additionally, the UINT32 flags MUST be set to zero.

## 1254**5. Endorsement Key Creation**

### 1255**Start of informative comment**

1256The TPM contains a 2048-bit RSA key pair called the endorsement key (EK). The public  
1257portion of the key is the PUBEK and the private portion the PRIVEK. Due to the nature of  
1258this key pair, both the PUBEK and the PRIVEK have privacy and security concerns.

1259The TPM has the EK generated before the end customer receives the platform. The Trusted  
1260Platform Module Entity (TPME) that causes EK generation is also the entity that will create  
1261and sign the EK credential attesting to the validity of the TPM and the EK. The TPME is  
1262typically the TPM manufacturer.

1263The TPM can generate the EK internally using the TPM\_CreateEndorsementKey or by using  
1264an outside key generator. The EK needs to indicate the genealogy of the EK generation.

1265Subsequent attempts to either generate an EK or insert an EK must fail.

1266If the data structure TPM\_ENDORSEMENT\_CREDENTIAL is stored on a platform after an  
1267Owner has taken ownership of that platform, it SHALL exist only in storage to which access  
1268is controlled and is available to authorized entities.

### 1269**End of informative comment**

12701. The EK MUST be a 2048-bit RSA key

1271 a. The public portion of the key is the PUBEK

1272 b. The private portion of the key is the PRIVEK

1273 c. The PRIVEK SHALL exist only in a TPM-shielded location.

12742. Access to the PRIVEK and PUBEK MUST only be via TPM protected capabilities

1275 a. The protected capabilities MUST require TPM Owner authentication or operator  
1276 physical presence

12773. The generation of the EK may use a process external to the TPM and  
1278 TPM\_CreateEndorsementKeyPair

1279 a. The external generation MUST result in an EK that has the same properties as an  
1280 internally generated EK

1281 b. The external generation process MUST protect the EK from exposure during the  
1282 generation and insertion of the EK

1283 c. After insertion of the EK the TPM state MUST be the same as the result of the  
1284 TPM\_CreateEndorsementKeyPair execution

1285 d. The process MUST guarantee correct generation, cryptographic strength,  
1286 uniqueness, privacy, and installation into a genuine TPM, of the EK

1287 e. The entity that signs the EK credential MUST be satisfied that the generation process  
1288 properly generated the EK and inserted it into the TPM

1289 f. The process MUST be defined in the target of evaluation (TOE) of the security target  
1290 in use to evaluate the TPM

## 1291**5.1 Controlling Access to PRIVEK**

### 1292**Start of informative comment**

1293Exposure of the PRIVEK is a security concern.

1294The TPM must ensure that the PRIVEK is not exposed outside of the TPM

### 1295**End of informative comment**

12961. The PRIVEK MUST never be out of the control of a TPM shielded location

## 1297**5.2 Controlling Access to PUBEK**

### 1298**Start of informative comment**

1299There are no security concerns with exposure or use of the PUBEK.

1300Privacy guidelines suggest that PUBEK could be considered personally identifiable  
1301information (PII) if it were associated in some way with personal information (PI) or  
1302associated with other PII, but PUBEK alone cannot be considered PII. Arbitrary random  
1303numbers do not represent a threat to privacy unless further associated with PI or PII. The  
1304PUBEK is an arbitrary random number that may be associated with aggregate platform  
1305information, but not personally identifiable information.

1306An EK may become associated with personally identifiable information when an alias  
1307platform identifier (AIK) is also associated with PI. The attestation service could include  
1308personal information in the AIK credential, thereby making the AIK-PUBEK association PII –  
1309but not before.

1310The association of PUBEK with AIK therefore is important to protect via privacy guidelines.  
1311The owner/user of the TPM should be able to control whether PUBEK is disclosed along  
1312with AIK. The owner/user should be notified of personal information that might be added to  
1313an AIK credential, which could result in AIK being considered PII. The owner/user should  
1314be able to evaluate the mechanisms used by an attestation entity to protect PUBEK-AIK  
1315associations before disclosure occurs. No other entity should be privy to owner/user  
1316authorized disclosure besides the intended attestation entity.

1317Several commands may be used to negotiate the conditions of PUBEK-AIK disclosure.  
1318TPM\_MakeIdentity discloses PUBEK-AIK in the context of requesting an AIK credential.  
1319TPM\_ActivateIdentity ensures the owner/user has not been spoofed by an interloper. These  
1320interfaces allow the owner/user to choose whether disclosure is acceptable and control the  
1321circumstances under which disclosure takes place. They do not allow the owner/user the  
1322ability to retain control of PUBEK-AIK subsequent to disclosure except by traditional means  
1323of trusting the attestation entity to abide by an acceptable privacy policy. The owner/user is  
1324able to associate the accepted privacy policy with the disclosure operation (e.g.  
1325TPM\_MakeIdentity).

1326A persistent flag called readPubek can be set to TRUE to permit reading of PUBEK via  
1327TPM\_ReadPubek. Reporting the PUBEK value is not considered privacy sensitive because it  
1328cannot be associated with any of the AIK keys managed by the TPM without using TPM  
1329protected-capabilities. Keys are encrypted with a nonce when flushed from TPM shielded-  
1330locations, Cryptanalysis of flushed keys will not reveal an association of EK to any AIK.

1331The command that manipulates the readPubek flag is TPM\_DisablePubekRead.

### 1332**End of informative comment**

## 1333 **6. Attestation Identity Keys**

1334 **Start of informative comment**

1335 See 11.4 Attestation Identity Keys.

1336 **End of informative comment**

## 1337 **7. TPM Ownership**

### 1338 **Start of informative comment**

1339 Taking ownership of a TPM is the process of inserting a shared secret into a TPM shielded-  
1340 location. Any entity that knows the shared secret is a TPM Owner. Proof of ownership  
1341 occurs when an entity, in response to a challenge, proves knowledge of the shared secret.  
1342 Certain operations in the TPM require authentication from a TPM Owner.

1343 Certain operations also allow the human, with physical possession of the platform, to assert  
1344 TPM Ownership rights. When asserting TPM Ownership, using physical presence, the  
1345 operations must not expose any secrets protected by the TPM.

1346 The platform owner controls insertion of the shared secret into the TPM. The platform  
1347 owner sets the NV persistent flag ownershipEnabled that allows the execution of the  
1348 TPM\_TakeOwnership command. The TPM\_SetOwnerInstall, the command that controls the  
1349 value ownershipEnabled, requires the assertion of physical presence.

1350 Attempting to execute TPM\_TakeOwnership fails when a TPM already has an owner. To  
1351 remove an owner when the current TPM Owner is unable to remove themselves, the human  
1352 that is in possession of the platform asserts physical presence and executes  
1353 TPM\_ForceClear which removes the shared secret.

1354 The insertion protocol that supplies the shared secret has the following requirements:  
1355 confidentiality, integrity, remoteness and verifiability.

1356 To provide confidentiality the proposed TPM Owner encrypts the shared secret using the  
1357 PUBEK. This requires the PRIVEK to decrypt the value. As the PRIVEK is only available in  
1358 the TPM the encrypted shared secret is only available to the intended TPM.

1359 The integrity of the process occurs by the TPM providing proof of the value of the shared  
1360 secret inserted into the TPM.

1361 By using the confidentiality and integrity, the protocol is useable by TPM Owners that are  
1362 remote to the platform.

1363 The new TPM Owner validates the insertion of the shared secret by using integrity response.

### 1364 **End of informative comment**

1365 The TPM MUST ship with no Owner installed. The TPM MUST use the ownership-control  
1366 protocol (OIAP or OSAP)

## 1367 **7.1 Platform Ownership and Root of Trust for Storage**

### 1368 **Start of informative comment**

1369 The semantics of platform ownership are tied to the Root-of-trust-for-storage (RTS). The  
1370 TPM\_TakeOwnership command creates a new Storage Root Key (SRK) and new tpmProof  
1371 value whenever a new owner is established. It follows that objects owned by a previous  
1372 owner will not be inherited by the new owner. Objects that should be inherited must be  
1373 transferred by deliberate data migration actions.

### 1374 **End of informative comment**

## 13758. Authentication and Authorization Data

### 1376Start of informative comment

1377Using security vernacular the terms below apply to the TPM for this discussion:

1378 Authentication: The process of providing proof of claimed ownership of an object or a  
1379subject's claimed identity.

1380 Authorization: Granting a subject appropriate access to an object.

1381Each TPM object that does not allow "public" access contains a 160-bit shared secret. This  
1382shared secret is enveloped within the object itself. The TPM grants use of TPM objects based  
1383on the presentation of the matching 160-bits using protocols designed to provide protection  
1384of the shared secret. This shared secret is called the AuthData.

1385Neither the TPM, nor its objects (such as keys), contain access controls for its objects (the  
1386exception to this is what is provided by the delegation mechanism). If an subject presents  
1387the AuthData, that subject is granted full use of the object based on the object's  
1388capabilities, not a set of rights or permissions of the subject. This apparent overloading of  
1389the concepts of authentication and authorization has caused some confusion. This is  
1390caused by having two similarly rooted but distinct perspectives.

1391From the perspective of the TPM looking out, this AuthData is its sole mechanism for  
1392authenticating the owner of its objects, thus from its perspective it is authentication data.  
1393However, from the application's perspective this data is typically the result of other  
1394functions that might perform authentications or authorizations of subjects using higher  
1395level mechanisms such as OS login, file system access, etc. Here, AuthData is a result of  
1396these functions so in this usage, it authorizes access to the TPM's objects. From this  
1397perspective, i.e., the application looking in on the TPM and its objects, the AuthData is  
1398authorization data. For this reason, and thanks to a common root within the English  
1399language, the term for this data is chosen to be AuthData and is to be interpreted or  
1400expanded as either authentication data or authorization data depending on context and  
1401perspective.

1402The term AuthData refers to the 160-bit value used to either prove ownership of, or  
1403authorization to use, an object. This is also called the object's shared secret. The term  
1404authorization will be used when referring the combined action of verifying the AuthData and  
1405allowing access to the object or function. The term authorization session applies to a state  
1406where the AuthData has been authentication and a session handle established that is  
1407associated with that authentication.

1408A wide-range of objects use AuthData. It is used to establish platform ownership, key use  
1409restrictions, object migration and to apply access control to opaque objects protected by the  
1410TPM.

1411AuthData is a 160-bit shared-secret plus high-entropy random number. The assumption is  
1412the shared-secret and random number are mixed using SHA-1 digesting, but no specific  
1413function for generating AuthData is specified by TCG.

1414TCG command processing sessions (e.g. OSAP, ADIP) may use AuthData as an initialization  
1415vector when creating a one-time pad. Session encryption is used to encrypt portions of  
1416command messages exchanged between TPM and a caller.

1417 The TPM stores AuthData with TPM controlled-objects and in shielded-locations. AuthData  
1418 is never in the clear, when managed by the TPM except in shielded-locations. Only TPM  
1419 protected-capabilities may access AuthData (contained in the TPM). AuthData objects may  
1420 not be used for any other purpose besides authentication and authorization of TPM  
1421 operations on controlled-objects.

1422 Outside the TPM, a reference monitor of some kind is responsible for protecting AuthData.  
1423 AuthData should be regarded as a controlled data item (CDI) in the context of the security  
1424 model governing the reference monitor. TCG expects this entity to preserve the interests of  
1425 the platform Owner.

1426 There is no requirement that instances of AuthData be unique.

#### 1427 **End of informative comment**

1428 The TPM MUST reserve 160 bits for the AuthData. The TPM treats the AuthData as a blob.  
1429 The TPM MUST keep AuthData in a shielded-location.

1430 The TPM MUST enforce that the only usage in the TPM of the AuthData is to perform  
1431 authorizations.

## 1432 **8.1 Dictionary Attack Considerations**

### 1433 **Start of informative comment**

1434 The decision to provide protections against dictionary attacks is due to the inability of the  
1435 TPM to guarantee that an authorization value has high entropy. While the creation and  
1436 authorization protocols could change to support the assurance of high entropy values, the  
1437 changes would be drastic and would totally invalidate any 1.x TPM version.

1438 Version 1.1 explicitly avoided any requirements for dictionary attack mitigation.

1439 Version 1.2 adds the requirement that the TPM vendor provide some assistance against  
1440 dictionary attacks. The internal mechanism is vendor specific. The TPM designer should  
1441 review the requirements for dictionary attack mitigation in the Common Criteria.

1442 The 1.2 specification does not provide any functions to turn on the dictionary attack  
1443 prevention. The specification does provide a way to reset from the TPM response to an  
1444 attack.

1445 By way of example, the following is a way to implement the dictionary attack mitigation.

1446 The TPM keeps a count of failed authorization attempts. The vendor allows the TPM Owner  
1447 to set a threshold of failed authorizations. When the count exceeds the threshold, the TPM  
1448 locks up and does not respond to any requests for a time out period. The time out period  
1449 doubles each time the count exceeds the threshold. If the TPM resets during a time out  
1450 period, the time out period starts over after TPM\_Init, or TPM\_Startup. To reset the count  
1451 and the time out period the TPM Owner executes TPM\_ResetLockValue. If the authorization  
1452 for TPM\_ResetLockValue fails, the TPM must lock up for the entire time out period and no  
1453 additional attempts at unlocking will be successful. Executing TPM\_ResetLockValue when  
1454 outside of a time out period still results in the resetting of the count and time out period.

### 1455 **End of informative comment**

1456 The TPM SHALL incorporate mechanism(s) that will provide some protection against  
1457 exhaustive or dictionary attacks on the authorization values stored within the TPM.



1458This version of the TPM specification does NOT specify the particular strategy to be used.  
1459Some examples might include locking out the TPM after a certain number of failures,  
1460forcing a reboot under some combination of failures, or requiring specific actions on the  
1461part of some actors after an attack has been detected. The mechanisms to manage these  
1462strategies are vendor specific at this time.

1463If the TPM in response to the attacks locks up for some time period or requires a special  
1464operation to restart, the TPM MUST prevent any authorized TPM command and MAY  
1465prevent any TPM command from executing until the mitigation mechanism completes. The  
1466TPM Owner can reset the mechanism using the TPM\_ResetLockValue command.  
1467TPM\_ResetLockValue MUST be allowed to run exactly once while the TPM is locked up.

## 1468 9. TPM Operation

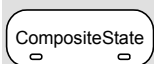
### 1469 Start of informative comment

1470 Through the course of TPM operation, it may enter several operational modes that include  
1471 power-up, self-test, administrative modes and full operation. This section describes TPM  
1472 operational states and state transition criteria. Where applicable, the TPM commands used  
1473 to facilitate state transition or function are included in diagrams and descriptions.

1474 The TPM keeps the information relative to the TPM operational state in a combination of  
1475 persistent and volatile flags. For ease of reading the persistent flags are prefixed by pFlags  
1476 and the volatile flags prefixed by vFlags.

1477 The following state diagram describes TPM operational states at a high level. Subsequent  
1478 state diagrams drill-down to finer detail that describes fundamental operations, protections  
1479 on operations and the transitions between them.

1480 The state diagrams use the following notation:



1481 - Signifies a state.



1482 - Transitions between states are represented as a single headed arrows.



1483 - Circular transitions indicate operations that don't result in a transition to another  
1484 state.



1485 - Decision boxes split state flow based on a logical test. Decision conditions are called  
1486 Guards and are identified by bracketed text.

1487 < [text] > Bracketed text indicates transitions that are gated. Text within the brackets  
1488 describes the pre-condition that must be met before state transition may occur.

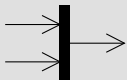
1489 < /name > Transitions may list the events that trigger state transition. The forward slash  
1490 demarcates event names.



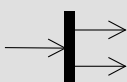
1491 - The starting point for reading state diagrams.



1492 - The ending point for state diagrams. Perpetual state systems may not have an ending  
1493 indicator.



1494 - The collection bar consolidates multiple identical transition events into a single  
1495 transition arrow.



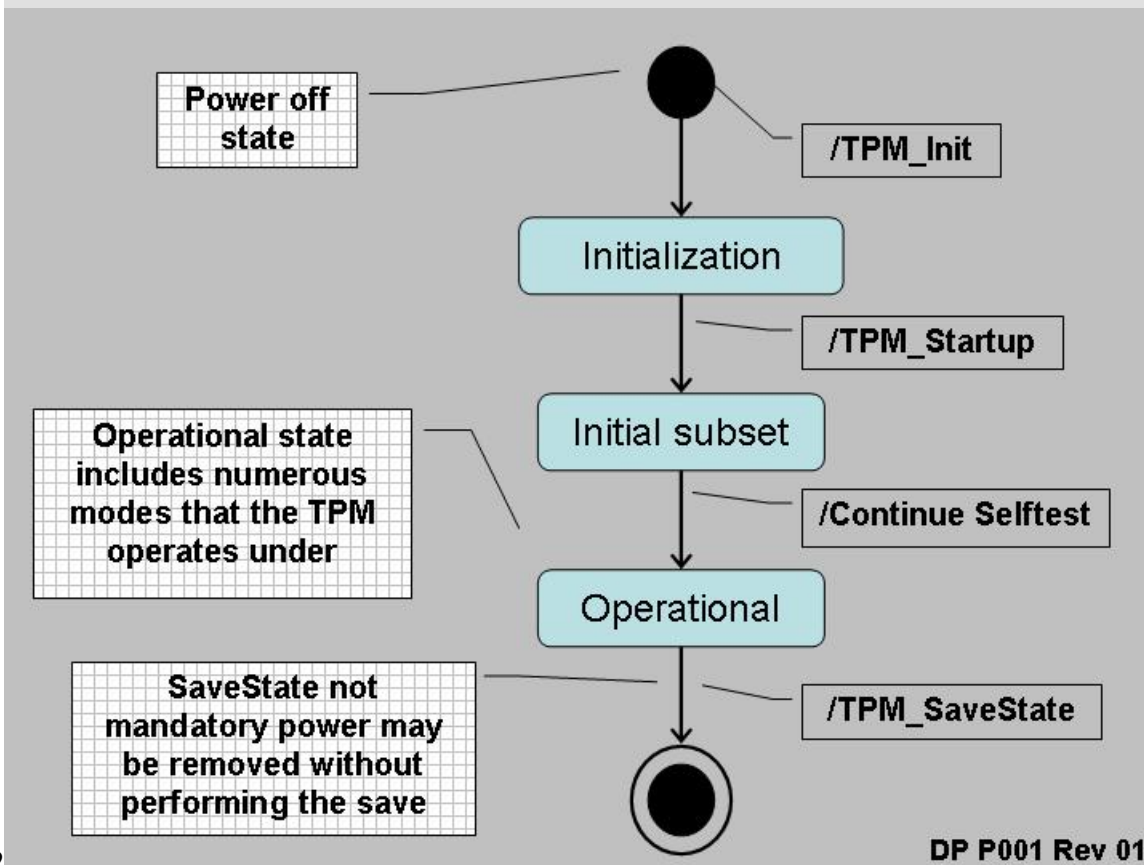
1496 - The distribution bar splits transitions to flow into multiple states.

1497 **(H)** - The history indicator means state values are remembered across context switches or  
 1498power-cycles.

1499**End of informative comment**

## 1500 9.1 TPM Initialization & Operation State Flow

1501**Start of informative comment**



1502  
 1503**Figure 9:b - TPM Operational States**

1504**End of informative comment**

### 1505 9.1.1 Initialization

1506**Start of informative comment**

1507TPM\_Init transitions the TPM from a power-off state to one where the TPM begins an  
 1508initialization process. TPM\_Init could be the result of power being applied to the platform or  
 1509a hard reset.

1510TPM\_Init sets an internal flag to indicate that the TPM is undergoing initialization. The TPM  
 1511must complete initialization before it is operational. The completion of initialization requires  
 1512the receipt of the TPM\_Startup command.

1513The TPM is not fully operational until all of the self-tests are complete. Successful  
 1514completion of the self-tests allows the TPM to enter fully operational mode.

1515 Fully operational does not imply that all functions of the TPM are available. The TPM needs  
1516 to have a TPM Owner and be enabled for all functions to be available.

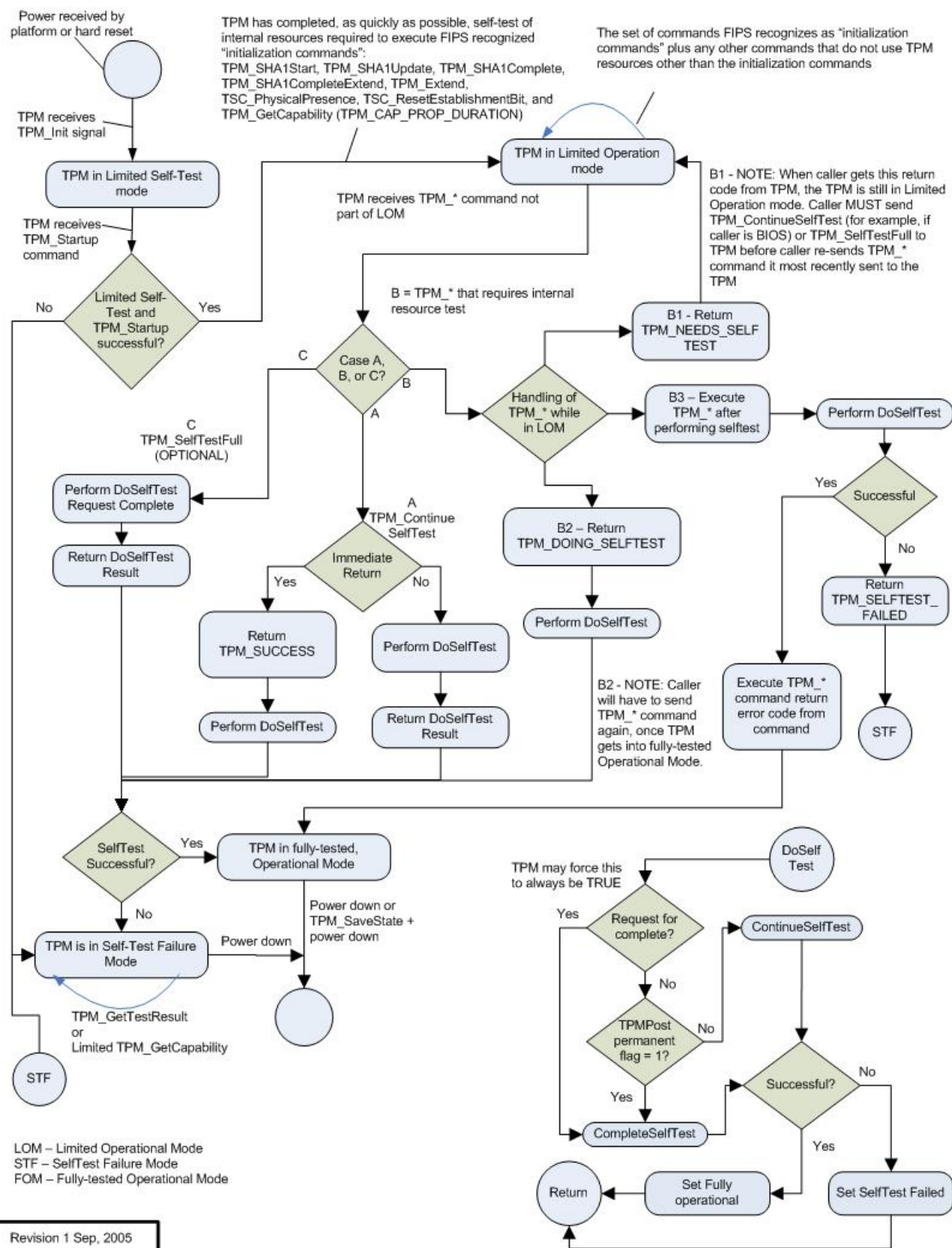
1517 The TPM transitions out of the operational mode by having power removed from the system.  
1518 Prior to the exiting operational mode, the TPM prepares for the transition by executing the  
1519 TPM\_SaveState command. There is no requirement that TPM\_SaveState execute before the  
1520 transition to power-off mode occurs.

1521 **End of informative comment**

1522 1. After TPM\_Init and until receipt of TPM\_Startup the TPM MUST return  
1523 TPM\_INVALID\_POSTINIT for all commands. Prior to receipt of TPM\_Startup the TPM  
1524 MAY enter shutdown or failure mode.

1525    **9.2       Self-Test Modes**

1526**Start of informative comment**



1529After initialization the TPM performs a limited self-test. This test provides the assurance  
1530that a selected subset of TPM commands will perform properly. The limited nature of the  
1531self-test allows the TPM to be functional in as short of time as possible. The commands  
1532enabled by this self-test are:

1533TPM\_SHA1xxx – Enabling the SHA-1 commands allows the TPM to assist the platform  
1534startup code. The startup code may execute in an extremely constrained memory  
1535environment and having the TPM resources available to perform hash functions can allow  
1536the measurement of code at an early time. While the hash is available, there are no speed  
1537requirements on the I/O bus to the TPM or on the TPM itself so use of this functionality  
1538may not meet platform startup requirements.

1539TPM\_Extend – Enabling the extend, and by reference the PCR, allows the startup code to  
1540perform measurements. Extending could use the SHA-1 TPM commands or perform the  
1541hash using the main processor.

1542TPM\_Startup – This command must be available as it is the transition command from the  
1543initial environment to the limited operational state.

1544TPM\_ContinueSelfTest – This command causes the TPM to complete the self-tests on all  
1545other TPM functions. If TPM receives a command, and the self-test for that command has  
1546not been completed, the TPM may implicitly perform the actions of the  
1547TPM\_ContinueSelfTest command.

1548TPM\_SelfTestFull – A TPM MAY allow this command after initialization, but typically  
1549TPM\_ContinueSelfTest would be used to avoid repeating the limited self tests.

1550TPM\_GetCapability – A subset of capabilities can be read in the limited operation state.

1551TSC\_PhysicalPresence and TSC\_ResetEstablishmentBit.

1552The complete self-test ensures that all TPM functionality is available and functioning  
1553properly.

#### 1554**End of informative comment**

15551. At startup, a TPM MUST self-test all internal functions that are necessary to do  
1556 TPM\_SHA1Start, TPM\_SHA1Update, TPM\_SHA1Complete, TPM\_SHA1CompleteExtend,  
1557 TPM\_Extend, TPM\_Startup, TPM\_ContinueSelfTest, a subset of TPM\_GetCapability,  
1558 TPM\_GetTestResult, TSC\_PhysicalPresence and TSC\_ResetEstablishmentBit.

15592. The TPM MAY allow TPM\_SelfTestFull to be used before completion of the actions of  
1560 TPM\_ContinueSelfTest.

15613. The TPM MAY implicitly run the actions of TPM\_ContinueSelfTest upon receipt of a  
1562 command that requires untested resources.

15634. The platform specific specification MUST define the maximum startup self-test time.

## 1564 **9.2.1 Operational Self-Test**

### 1565**Start of informative comment**

1566The completion of self-test is initiated by TPM\_ContinueSelfTest. The TPM MAY allow  
1567TPM\_SelfTestFull to be issued instead of TPM\_ContinueSelfTest.

1568TPM\_ContinueSelfTest is the command issued during platform initialization after the  
1569platform has made use of the early commands (perhaps for an early measurement), the

1570 platform is now performing other initializations, and the TPM can be left alone to complete  
1571 the self-tests. Before any command other than the limited subset is executed, all self-tests  
1572 must be complete.

1573 TPM\_SelfTestFull is a request to have the TPM perform another complete self-test. This test  
1574 will take some time but provides an accurate assessment of the TPM's ability to perform all  
1575 operations.

1576 The original design of TPM\_ContinueSelfTest was for the TPM to test those functions that  
1577 the original startup did not test. The FIPS-140 evaluation of the specification requested a  
1578 change such that TPM\_ContinueSelfTest would perform a complete self-test. The rationale  
1579 is that the original tests are only part of the initialization of the TPM; if they fail, the TPM  
1580 does not complete initialization. Performing a complete test after initialization meets the  
1581 FIPS-140 requirements. The TPM may work differently in FIPS mode or the TPM may simply  
1582 write the TPM\_ContinueSelfTest command such that it always performs the complete check.

1583 TPM\_ContinueSelfTest causes a test of the TPM internal functions. When  
1584 TPM\_ContinueSelfTest is asynchronous, the TPM immediately returns a successful result  
1585 code before starting the tests. When testing is complete, the TPM does not return any  
1586 result. When TPM\_ContinueSelfTest is synchronous, the TPM completes the self-tests and  
1587 then returns a success or failure result code.

1588 The TPM may reject any command other than the limited subset if self test has not been  
1589 completed. Alternatively, the actions of TPM\_ContinueSelfTest may start automatically if the  
1590 TPM receives a command and there has been no testing of the underlying functionality. If  
1591 the TPM implements this implicit self-test, it may immediately return a result code  
1592 indicating that it is doing self-test. Alternatively, it may do the self-test, then do the  
1593 command, and return only the result code of the command.

1594 Programmers of TPM drivers should take into account the time estimates for self-test and  
1595 minimize the polling for self-test completion. While self-test is executing, the TPM may  
1596 return an out-of-band "busy" signal to prevent command from being issued. Alternatively,  
1597 the TPM may accept the command but delay execution until after the self-test completes.  
1598 Either of those alternatives may appear as if the TPM is blocking to upper software layers.  
1599 Alternatively, the TPM may return an indication that is doing a self-test.

1600 Upon the completion of the self-tests, the result of the self-tests are held in the TPM such  
1601 that a subsequent call to TPM\_GetTestResult returns the self-test result.

1602 In version 1.1, there was a separate command to create a signed self-test,  
1603 TPM\_CertifySelfTest. Version 1.2 deprecates the command. The new use model is to perform  
1604 TPM\_GetTestResult inside of a transport session and then use  
1605 TPM\_ReleaseTransportSigned to obtain the signature.

1606 If self-tests fail, the TPM goes into failure state and does not allow most other operations to  
1607 continue. The TPM\_GetTestResult will operate in failure mode so an outside observer can  
1608 obtain information as to the reason for the self-test failure.

1609 A TPM may take three courses of action when presented with a command that requires an  
1610 untested resource.

1611 1. The TPM may return TPM\_NEEDS\_SELFTEST, indicating that the execution of the  
1612 command requires TPM\_ContinueSelfTest.



254

16132. The TPM may implicitly execute the self-test and return a TPM\_DOING\_SELFTEST  
1614 return code, causing the external software to retry the command.

16153. The TPM may implicitly execute the self-test, execute the ordinal, and return the results  
1616 of the ordinal.

1617The following example shows how software can detect either mechanism with a single piece  
1618of code

16191. SW sends TPM\_xxx command

16202. SW checks return code from TPM

16213. If return code is TPM\_DOING\_SELFTEST, SW attempts to resend

1622 a. If the TIS times out waiting for TPM ready, pause for self-test time then resend

1623 b. if TIS timeout, then error

16244. else if return code is TPM\_NEEDS\_SELFTEST

1625 a. Send TPM\_ContinueSelfTest

16265. else

1627 a. Process the ordinal return code

1628**End of informative comment**

16291. The TPM MUST provide startup self-tests. The TPM MUST provide mechanisms to allow  
1630 the self-tests to be run on demand. The response from the self-tests is pass or fail.

16312. The TPM MUST complete the startup self-tests in a manner and timeliness that allows  
1632 the TPM to be of use to the BIOS during the collection of integrity metrics.

16333. The TPM MUST complete the required checks before a given feature is in use. If a  
1634 function self-test is not complete the TPM MUST return TPM\_NEEDS\_SELFTEST or  
1635 TPM\_DOING\_SELFTEST, or do the self-test before using the feature.

16364. There are two sections of startup self-tests: required and recommended. The  
1637 recommended tests are not a requirement due to time constraints. The TPM  
1638 manufacturer should perform as many tests as possible within the time constraints.

16395. The TPM MUST report the result of the tests that it performs.

16406. The TPM MUST provide a mechanism to allow self-test to execute on request by any  
1641 challenger.

16427. The TPM MUST provide for testing of some operations during each execution of the  
1643 operation.

16448. The TPM MUST check the following:

1645 a. RNG functionality

1646 b. Reading and extending the integrity registers. The self-test for the integrity registers  
1647 will leave the integrity registers in a known state.

1648 c. Testing the EK integrity, if it exists

- 1649 i. This requirement specifies that the TPM will verify that the endorsement key pair  
1650 can encrypt and decrypt a known value. This tests the RSA engine. If the EK has  
1651 not yet been generated the TPM action is manufacturer specific.
- 1652 d. The integrity of the protected capabilities of the TPM
- 1653 i. This means that the TPM must ensure that its “microcode” has not changed, and  
1654 not that a test must be run on each function.
- 1655 e. Any tamper-resistance markers
- 1656 i. The tests on the tamper-resistance or tamper-evident markers are under  
1657 programmable control. There is no requirement to check tamper-evident tape or  
1658 the status of epoxy surrounding the case.

16599. The TPM MUST check the following:

- 1660 a. The hash functionality
- 1661 i. This check MAY hash a known value and compare it to an expected result.
- 1662 b. Any symmetric algorithms
- 1663 i. This check MAY use known data with a random key to encrypt and decrypt the  
1664 data
- 1665 c. Any asymmetric algorithms
- 1666 i. This check MAY use known data to encrypt and decrypt.
- 1667 d. Any hardware crypto accelerators

#### 1668 10. Self-Test Failure

- 1669 a. When the TPM detects a failure during any self-test, the TPM MUST enter shutdown  
1670 mode. This shutdown mode will allow only the following operations to occur:
- 1671 i. Update. The update function MAY replace invalid microcode, providing that the  
1672 parts of the TPM that provide update functionality have passed self-test.
- 1673 ii. TPM\_GetTestResult. This command can assist the TPM manufacturer in  
1674 determining the cause of the self-test failure.
- 1675 iii. TPM\_GetCapability may return limited information as specified in the ordinal.
- 1676 iv. All other operations will return the error code TPM\_FAILEDSELFTEST.
- 1677 b. The TPM MUST leave failure mode only after receipt of TPM\_Init.
- 1678 c. When the TPM detects a failure during any self-test, it SHOULD delete values  
1679 preserved by TPM\_SaveState.

1680 11. Prior to the completion of the actions of TPM\_ContinueSelfTest the TPM MAY respond in  
1681 two ways

- 1682 a. The TPM MAY automatically invoke the actions of TPM\_ContinueSelfTest.
- 1683 i. The TPM MAY return TPM\_DOING\_SELFTEST.
- 1684 ii. The TPM may complete the self-test, execute the command, and return the  
1685 command result.
- 1686 b. The TPM MAY return the error code TPM\_NEEDS\_SELFTEST

1687 **9.3 Startup**1688 **Start of informative comment**

1689Startup transitions the TPM from the initialization state to an operational state. The  
1690transition includes information from the platform to inform the TPM of the platform  
1691operating state. TPM\_Startup has three options: Clear, State and Deactivated.

1692The Clear option informs the TPM that the platform is starting in a “cleared” state or most  
1693likely a complete reboot. The TPM is to set itself to the default values and operational state  
1694specified by the TPM Owner.

1695The State option informs the TPM that the platform is requesting the TPM to recover a saved  
1696state and continue operation from the saved state. The platform previously made the  
1697TPM\_SaveState request to the TPM such that the TPM prepares values to be recovered later.  
1698If the TPM enters failure mode after TPM\_SaveState, the saved state should be deleted. It is  
1699then possible that the State option will fail.

1700The Deactivated state informs the TPM that it should not allow further operations and  
1701should fail all subsequent command requests. The Deactivated state can only be reset by  
1702performing another TPM\_Init.

1703 **End of informative comment**1704 **9.4 Operational Mode**1705 **Start of informative comment**

1706After the TPM completes both TPM\_Startup and self-tests, the TPM is ready for operation.

1707There are three discrete states, enabled or disabled, active or inactive and owned or  
1708unowned. These three states when combined form eight operational modes.



DP P003 Rev 01

1709

1710Figure 9:d - Eight Modes of Operation

1711S1 is the fully operational state where all TPM functions are available. S8 represents a mode  
1712where all TPM features (except those to change the state) are off.

1713Given the eight modes of operation, the TPM can be flexible in accommodating a wide range  
1714of usage scenarios. The default delivery state for a TPM should be S8 (disabled, inactive and  
1715unowned). In S8, the only mechanism available to move the TPM to S1 is having physical  
1716access to the platform.

1717Two examples illustrate the possibilities of shipping combinations.

#### 1718Example 1

1719The customer does not want the TPM to attest to any information relative to the platform.  
1720The customer does not want any remote entity to attempt to change the control options that  
1721the platform owner is setting. For this customer the platform manufacturer sets the TPM in  
1722S8 (disabled, deactivated and unowned).

1723To change the state of the platform the platform owner would assert physical presence and  
1724enable, activate and insert the TPM Owner shared secret. The details of how to change the  
1725various modes is in subsequent sections.

1726This particular sequence gives maximum control to the customer.

#### 1727Example 2

1728A corporate customer wishes to have platforms shipped to their employees and the IT  
1729department wishes to take control of the TPM remotely. To satisfy these needs the TPM  
1730should be in S5 (enabled, active and unowned). When the platform connects to the  
1731corporate LAN the IT department would execute the TPM\_TakeOwnership command  
1732remotely.

1733This sequence allows the IT department to accept platforms into their network without  
1734having to have physical access to each new machine.

#### 1735**End of informative comment**

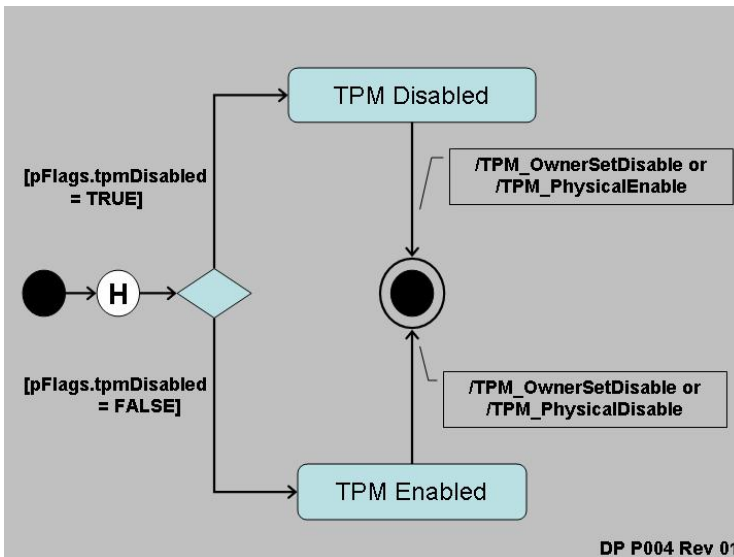
1736The TPM MUST have commands to perform the following:

- 17371. Enable and disable the TPM. These commands MUST work as TPM Owner authorized or  
1738 with the assertion of physical presence
- 17392. Activate and deactivate the TPM. These commands MUST work as TPM Owner  
1740 authorized or with the assertion of physical presence
- 17413. Activate and deactivate the ability to take ownership of the TPM
- 17424. Assert ownership of the TPM.

### 1743 **9.4.1 Enabling a TPM**

#### 1744**Informative comment**

1745A disabled TPM is not able to execute commands that use the resources of a TPM. While  
1746some commands are available (SHA-1 for example) the TPM is not able to load keys and  
1747perform TPM\_Seal and other such operations. These restrictions are the same as for an  
1748inactive TPM. The difference between inactive and disabled is that a disabled TPM is unable  
1749to execute the TPM\_TakeOwnership command. A disabled TPM that has a TPM Owner is not  
1750able to execute normal TPM commands.



1751  
1752pFlags.tpmDisabled contains the current enablement status. When set to TRUE the TPM is  
1753disabled, when FALSE the TPM is enabled.

1754Changing the setting pFlags.tpmDisabled has no effect on any secrets or other values held  
1755by the TPM. No keys, monotonic counters or other resources are invalidated by changing  
1756TPM enablement. There is no guarantee that session resources (like transport sessions)  
1757survive the change in enablement, but there is no loss of secrets.

1758The TPM\_OwnerSetDisable command can be used to transition in either Enabled or  
1759Disabled states. The desired state is a parameter to TPM\_OwnerSetDisable. This command  
1760requires TPM Owner authentication to operate. It is suitable for post-boot and remote  
1761invocation.

1762An unowned TPM requires the execution of TPM\_PhysicalEnable to enable the TPM and  
1763TPM\_PhysicalDisable to disable the TPM. Operators of an owned TPM can also execute  
1764these two commands. The use of the physical commands allows a platform operator to  
1765disable the TPM without TPM Owner authorization.

1766TPM\_PhysicalEnable transitions the TPM from Disabled to Enabled state. This command is  
1767guarded by a requirement of operator physical presence. Additionally, this command can be  
1768invoked by a physical event at the platform, whether or not the TPM has an Owner or there  
1769is a human physically present. This command is suitable for pre-boot invocation.

1770TPM\_PhysicalDisable transitions the TPM from Enabled to Disabled state. It has the same  
1771guard and invocation properties as TPM\_PhysicalEnable.

1772The subset of commands the TPM is able to execute is defined in the structures document  
1773in the persistent flag section.

1774Misuse of the disabled state can result in denial-of-service. Proper management of Owner  
1775AuthData and physical access to the platform is a critical element in ensuring availability of  
1776the system.

1777**End of informative comment**

17781. The TPM MUST provide an enable and disable command that is executed with TPM  
1779Owner authorization.

17802. The TPM MUST provide an enable and disable command this is executed locally using  
1781 physical presence.

## 1782 9.4.2 Activating a TPM

### 1783 Informative comment

1784A deactivated TPM is not able to execute commands that use TPM resources. A major  
1785difference between deactivated and disabled is that a deactivated TPM CAN execute the  
1786TPM\_TakeOwnership command.

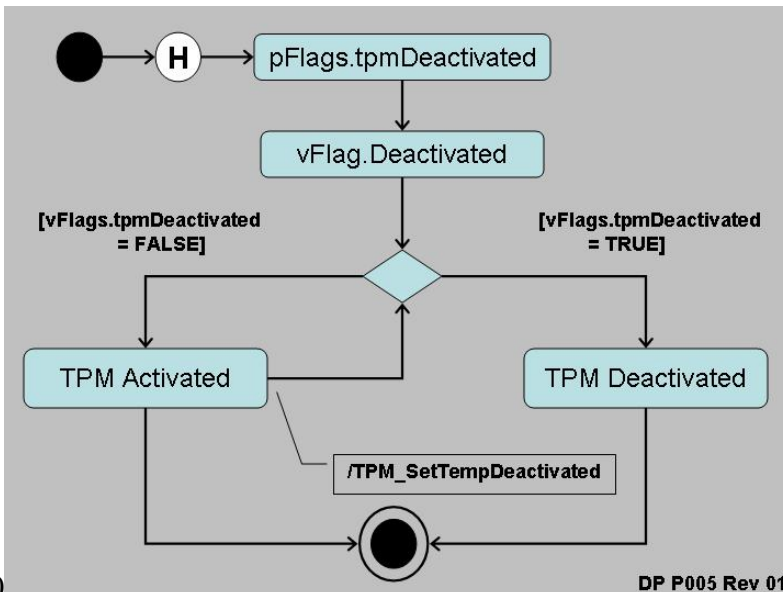
1787Deactivated may be used to prevent the (obscure) attack where a TPM is readied for  
1788TPM\_TakeOwnership but a remote rogue manages to take ownership of a platform just  
1789before the genuine owner, and immediately has use of the TPM's facilities. To defeat this  
1790attack, a genuine owner should set disable==FALSE, ownership==TRUE, deactivate==TRUE,  
1791execute TPM\_takeOwnership, and then set deactivate==FALSE after verifying that the  
1792genuine owner is the actual TPM owner.

1793Activation control is with both persistent and volatile flags. The persistent flag is never  
1794directly checked by the TPM, rather it is the source of the original setting for the volatile  
1795flag. During TPM initialization the value of pFlags.tpmDeactivated is copied to  
1796vFlags.tpmDeactivated. When the TPM execution engine checks for TPM activation, it only  
1797references vFlags.tpmDeactivated.

1798Toggling the state of pFlags.tpmDeactivated uses TPM\_PhysicalSetDeactivated. This  
1799command requires physical presence. There is no associated TPM Owner authenticated  
1800command as the TPM Owner can always execute TPM\_OwnerSetDisabled which results in  
1801the same TPM operations. The toggling of this flag does not affect the current operation of  
1802the TPM but requires a reboot of the platform such that the persistent flag is again copied  
1803to the volatile flag.

1804The volatile flag, vFlags.tpmDeactivated, is set during initialization by the value of  
1805pFlags.tpmDeactivated. If vFlags.tpmDeactivated is TRUE the only way to reactivate the  
1806TPM is to reboot the platform and have pFlags reset the vFlags value.

1807If vFlags.tpmDeactivated is FALSE, running TPM\_SetTempDeactivated will set  
1808vFlags.tpmDeactivated to TRUE and then require a reboot of the platform to reactivate the  
1809platform.



1810

1811Figure 9:e - Activated and Deactivated States

1812TPM activation is for Operator convenience. It allows the operator to deactivate the platform  
1813(temporarily, using TPM\_SetTempDeactivated) during a user session when the operator does  
1814not want to disclose platform or attestation identity. This provides operator privacy, since  
1815PCRs could provide cryptographic proof of an operation. PCRs are inaccessible when a TPM  
1816is deactivated. They cannot be used for authorization, nor can they be read. The reboot  
1817required to activate a TPM also resets the PCRs.

1818The subset of commands that are available when the TPM is deactivated is contained in the  
1819structures document. The TPM\_TakeOwnership command is available when deactivated.  
1820The TPM\_Extend command is available when deactivated so that software (e.g. a BIOS) can  
1821run the command without the need to handle an error. The PCR extend operation is  
1822irrelevant, since the resulting PCR value cannot be used.

### 1823End of informative comment

18241. The TPM MUST maintain a non-volatile flag that indicates the activation state
18252. The TPM MUST provide for the setting of the non-volatile flag using a command that  
1826 requires physical presence
18273. The TPM MUST sets a volatile flag using the current setting of the non-volatile flag.
18284. The TPM MUST provide for a command that deactivates the TPM immediately
18295. The only mechanism to reactivate a TPM once deactivated is to power-cycle the system.

## 1830 9.4.3 Taking TPM Ownership

### 1831Start of informative comment

1832The owner of the TPM has ultimate control of the TPM. The owner of the TPM can enable or  
1833disable the TPM, create AIK and set policies for the TPM. The process of taking ownership  
1834must be a tightly controlled process with numerous checks and balances.

1835The protections around the taking of ownership include the enablement status, specific  
1836persistent flags and the assertion of physical presence.

1837 Control of the TPM revolves around knowledge of the TPM Owner authentication value.  
1838 Proving knowledge of authentication value proves the calling entity is the TPM Owner. It is  
1839 possible for more than one entity to know the TPM Owner authentication value.

1840 The TPM provides no mechanisms to recover a lost TPM Owner authentication value.

1841 Recovery from a lost or forgotten TPM Owner authentication value involves removing the old  
1842 value and installing a new one. The removal of the old value invalidates all information  
1843 associated with the previous value. Insertion of a new value can occur after the removal of  
1844 the old value.

1845 A disabled and inactive TPM that has no TPM Owner cannot install an owner.

1846 To invalidate the TPM Owner authentication value use either TPM\_OwnerClear or  
1847 TPM\_ForceClear.

1848 **End of informative comment**

1849 1. The TPM Owner authentication value MUST be a 160-bits

1850 2. The TPM Owner authentication value MUST be held in persistent storage

1851 3. The TPM MUST have no mechanisms to recover a lost TPM Owner authentication value

## 1852 **9.4.3.1 Enabling Ownership**

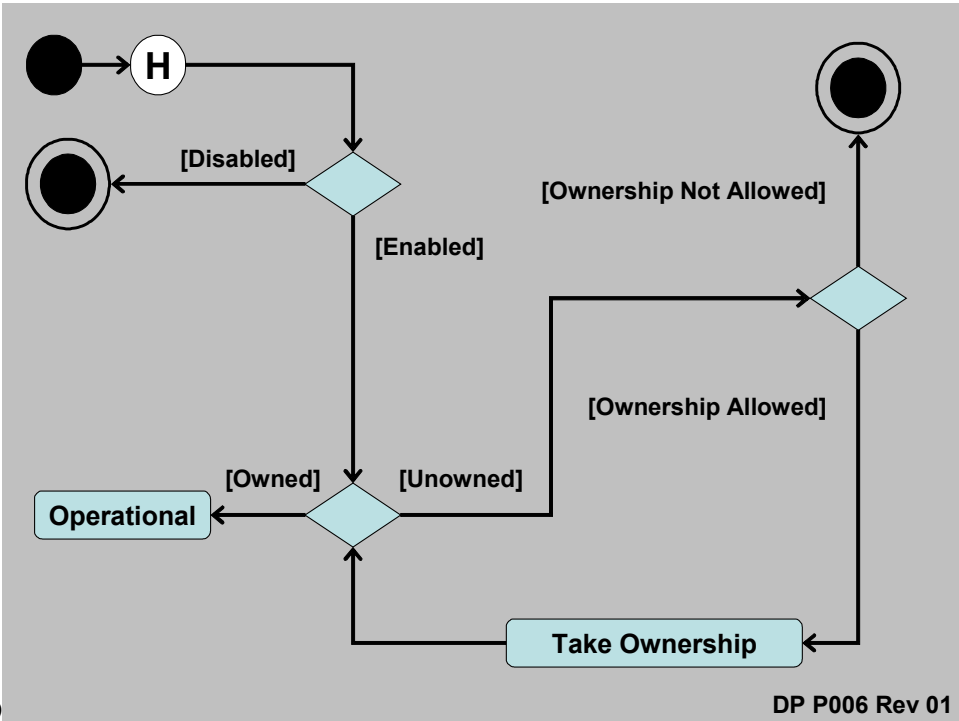
1853 **Informative comment**

1854 For the TPM\_TakeOwnership command to succeed, pFlags.disable must be FALSE and  
1855 pFlags.ownership must be TRUE.

1856 The following diagram shows the states and the operational checks the TPM makes before  
1857 allowing the insertion of the TPM Ownership value.

1858





DP P006 Rev 01

1859  
1860  
1861  
1862The TPM checks pFlags.disable. If the TPM is enabled, the TPM checks for the existence of  
1863a TPM Owner. If an Owner is not present, the TPM checks pFlags.ownership. If TRUE, the  
1864TPM\_TakeOwnership command will execute.  
1865While the TPM has no Owner but is enabled and active, a limited subset of commands will  
1866successfully execute.  
1867The TPM\_SetOwnerInstall command toggles the state of the pFlags.ownership flag.  
1868TPM\_SetOwnerInstall requires the assertion of physical presence to execute.  
1869**End of informative comment**

1870 **9.4.4 Transitioning Between Operational States**

1871**Start of informative comment**

1872The following table is a recap of the commands necessary to transition a TPM from one state  
1873to another.

State	TPM Owner Auth	Physical Presence	Persistence
Disabled to Enabled	TPM_OwnerSetDisable	TPM_PhysicalEnable	permanent
Enabled to Disabled	TPM_OwnerSetDisable	TPM_PhysicalDisable	permanent
Inactive to Active		TPM_PhysicalSetDeactivated	permanent
Active to Inactive		TPM_PhysicalSetDeactivated	permanent
Active to Inactive		TPM_SetTempDeactivated	boot cycle

1874

1875 **End of informative comment**

1876 **9.5 Clearing the TPM**

1877 **Start of informative comment**

1878 Clearing the TPM is the process of returning the TPM to factory defaults. It is possible the  
1879 platform owner will change when in this state.

1880 The commands to clear a TPM require either TPM Owner authentication or the assertion of  
1881 physical presence.

1882 The clear process performs the following tasks:

1883 Invalidate the SRK. Once invalidated all information stored using the SRK is now  
1884 unavailable. The invalidation does not change the blobs using the SRK rather there is no  
1885 way to decrypt the blobs after invalidation of the SRK.

1886 Invalidate tpmProof. tpmProof is a value that provides the uniqueness to values stored off of  
1887 the TPM. By invalidating tpmProof all off TPM blobs will no longer load on the TPM.

1888 Invalidate the TPM Owner authentication value. With the authentication value invalidated  
1889 there are no TPM Owner authenticated commands that will execute.

1890 Reset volatile and non-volatile data to manufacturer defaults.

1891 The clear must not affect the EK.

1892 Once cleared the TPM will return TPM\_NOSRK to commands that require authentication.

1893 The PCR values are undefined after a clear operation. The TPM must go through TPM\_Init to  
1894 properly set the PCR values.

1895 Clear authentication comes from either the TPM owner or the assertion of physical  
1896 presence. As the clear commands present a real opportunity for a denial of service attack  
1897 there are mechanisms in place disabling the clear commands.

1898 Disabling TPM\_OwnerClear uses the TPM\_DisableOwnerClear command. The state of ability  
1899 to execute TPM\_OwnerClear is then held as one of the non-volatile flags.

1900 Enablement of TPM\_ForceClear is held in the volatile disableForceClear flag.  
1901 disableForceClear is set to FALSE during TPM\_Init. To disable the command software  
1902 should issue the TPM\_DisableForceClear command.

1903 During the TPM startup processing anyone with physical access to the machine can issue  
1904 the TPM\_ForceClear command. This command performs the clear operations if it has not  
1905 been disabled by vFlags.DisabledForceClear being TRUE.

1906 The TPM can be configured to block all forms of clear operations. It is advisable to block  
1907 clear operations to prevent an otherwise trivial denial-of-service attack. The assumption is  
1908 the system startup code will issue the TPM\_DisableForceClear on each power-cycle after it  
1909 is determined the TPM\_ForceClear command will not be necessary. The purpose of the  
1910 TPM\_ForceClear command is to recover from the state where the Owner has lost or  
1911 forgotten the TPM Owner-authentication-data.

1912 The TPM\_ForceClear must only be possible when the issuer has physical access to the  
1913 platform. The manufacturer of a platform determines the exact definition of physical access.

1914The commands to clear a TPM require either TPM Owner authentication, TPM\_OwnerClear,  
1915or the assertion of physical presence, TPM\_ForceClear.

1916**End of informative comment**

19171. The TPM MUST support the clear operations.

1918 a. Clear operations MUST be authenticated by either the TPM Owner or physical  
1919 presence

1920 b. The TPM MUST support mechanisms to disable the clear operations

19212. The clear operation MUST perform at least the following actions

1922 a. SRK invalidation

1923 b. tpmProof invalidation

1924 c. TPM Owner authentication value invalidation

1925 d. Resetting non-volatile values to defaults

1926 e. Invalidation of volatile values

1927 f. Invalidation of internal resources

19283. The clear operation must not affect the EK.

## 10. Physical Presence

### Start of informative comment

This specification describes commands that require physical presence at the platform before the command will operate. Physical presence implies direct interaction by a person – i.e. Operator with the platform / TPM.

The type of controls that imply special privilege include:

- Clearing an existing Owner from the TPM,
- Temporarily deactivating a TPM,
- Temporarily disabling a TPM.

Physical presence implies a level of control and authorization to perform basic administrative tasks and to bootstrap management and access control mechanisms.

Protection of low-level administrative interfaces can be provided by physical and electrical methods; or by software; or a combination of both. The guiding principle for designers is the protection mechanism should be difficult or impossible to spoof by rogue software. Designers should take advantage of restricted states inherent in platform operation. For example, in a PC, software executed during the power-on self-test (POST) cannot be disturbed without physical access to the platform. Alternatively, a hardware switch indicating physical presence is very difficult to circumvent by rogue software or remote attackers.

TPM and platform manufacturers will determine the actual implementation approach. The strength of the protection mechanisms is determined by an evaluation of the platform.

Physical presence indication is implemented as a flag in volatile memory known as the physicalPresenceV flag. When physical presence is established (TRUE) several TPM commands are able to function. They include:

- TPM\_PhysicalEnable,
- TPM\_PhysicalDisable,
- TPM\_PhysicalSetDeactivated,
- TPM\_ForceClear,
- TPM\_SetOwnerInstall,

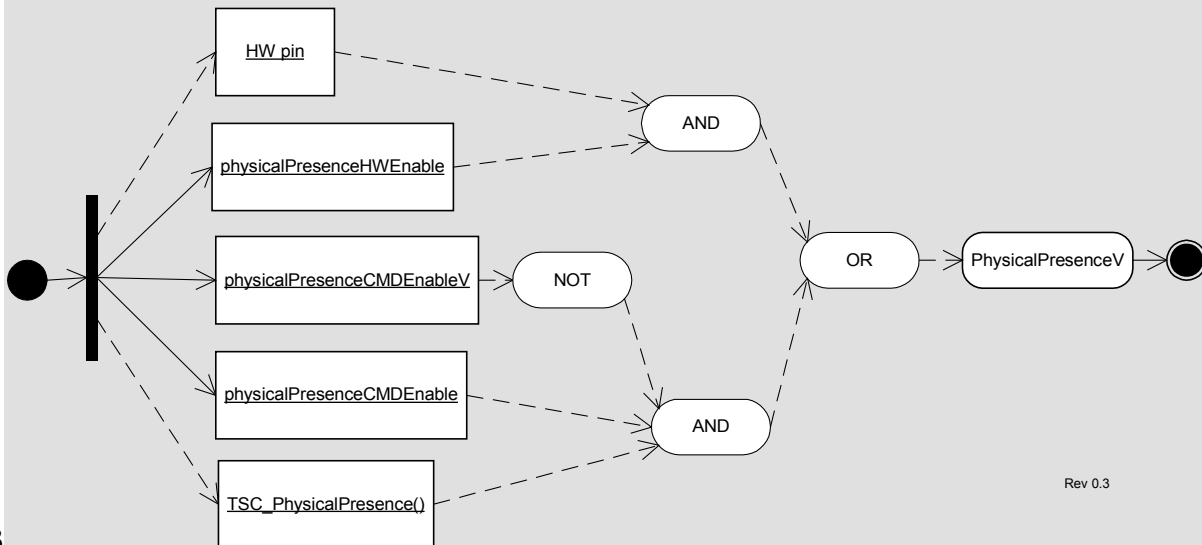
In order to execute these commands, the TPM must obtain unambiguous assurance that the operation is authorized by physical-presence at the platform. The command processor in the I/O component checks the physicalPresenceV flag before continuing processing of TPM command blocks. The volatile physicalPresenceV flag is set only while the Operator is indeed physically present.

TPM designers should take precautions to ensure testing of the physicalPresenceV flag value is not mask-able. For example, a special bus cycle could be used or a dedicated line implemented.

There is an exception to physical presence semantics that allows a remote entity the ability to assert physical presence when that entity is not physically present. The TSC\_PhysicalPresence command is used to change polarity of the physicalPresenceV flag.

1969Its use is heavily guarded. See sections describing the TPM Opt-In component; and Volatile  
1970and Non-volatile memory components.

1971The following diagram illustrates the flow of logic controlling updates to the  
1972physicalPresenceV flag:



1973

1974Figure 10:f - Physical Presence Control Logic

1975This diagram shows that the physicalPresenceV flag may be updated either by a HW pin or  
1976through the TSC\_PhysicalPresence command, but gated by persistent control flags and a  
1977temporal lock. Observe, the reverse logic surrounding the use of TSC\_PhysicalPresence  
1978command. When the physicalPresenceCmdEnable flag is set and the physicalPresenceLock  
1979flag is not set, the TSC\_PhysicalPresence command may set physicalPresenceV.

1980The physicalPresenceV flag may be overridden by unambiguous physical presence.  
1981Conceptually, the use of dedicated electrical hardware providing a trusted path to the  
1982Operator has higher precedence than the physicalPresenceV flag value. Implementers  
1983should consider this when implementing physical presence indicators.

#### 1984End of informative comment

19851. The requirement for physical presence MUST be met by the platform manufacturer  
1986 using some physical mechanism.

19872. It SHALL be impossible to intercept or subvert indication of physical presence to the  
1988 TPM by the execution of software on the platform.

## 11. Root of Trust for Reporting (RTR)

### Start of informative comment

The RTR is responsible for establishing platform identities, reporting platform configurations, protecting reported values, and providing a function for attesting to reported values. The RTR shares responsibility of protecting measurement digests with the RTS.

The interaction between the RTR and RTS is a critical component. The design and implementation of the interaction between the RTR and RTS should mitigate observation and tampering with the messages. It is strongly encouraged that the RTR and RTS implementation occur in the same package such there are no external observation points. For a silicon based TPM this would imply that the RTR and RTS are in the same silicon package with no external busses.

### End of informative comment

1. An instantiation of the RTS and RTR SHALL do the following:

- a. Be resistant to all forms of software attack and to the forms of physical attack implied by the platform's Protection Profile
- b. Supply an accurate digest of all sequences of presented integrity metrics

## 11.1 Platform Identity

### Start of informative comment

The RTR is a cryptographic identity used to distinguish and authenticate an individual TPM. The TPM uses the RTR to answer an integrity challenge.

In the TPM, the Endorsement Key (EK) is the RTR. The EK is cryptographically unique and bound to the TPM.

Prior to any use of the TPM, the RTR must be instantiated. Instantiation may occur during TPM manufacturing or platform manufacturing. The business issues and manufacturing flow determines how a specific TPM and platform is initialized with the EK.

As the RTR is cryptographically unique, the use of the RTR must only occur in controlled circumstances due to privacy concerns. The EK is only available for two operations: establishing the TPM Owner and establishing Attestation Identity Key (AIK) values and credentials. There is a prohibition on the use of the EK for any other operation.

### End of informative comment

1. The RTR MUST have a cryptographic identity.

- a. The cryptographic identity of the RTR is the Endorsement Key (EK).

2. The EK MUST be

- a. Statistically unique
  - i. When the TPM is in FIPS mode, the EK MUST be generated using a random number generator that meets FIPS requirements.
  - ii. Difficult to forge or counterfeit
- b. Verifiable during the AIK creation process

317

20273. The EK SHALL only participate in

2028 a. TPM Ownership insertion

2029 b. AIK creation and verification

## 2030 **11.2 RTR to Platform Binding**

### 2031 **Start of informative comment**

2032When performing validation of the EK and the platform the challenger wishes to have  
2033knowledge of the binding of RTR to platform. The RTR is bound to a TPM hence if the  
2034platform can show the binding of TPM to platform the challenger can reasonably believe the  
2035RTR and platform binding.

2036The TPM cannot provide all of the information necessary for the challenger to trust in the  
2037binding. That information comes from the manufacturing process and occurs outside the  
2038control of the TPM.

### 2039 **End of informative comment**

20401. The EK is transitively bound to the Platform via the TPM as follows:

2041 a. An EK is bound to one and only one TPM (i.e., there is a one to one correspondence  
2042 between an Endorsement Key and a TPM.)

2043 b. A TPM is bound to one and only one Platform. (i.e., there is a one to one  
2044 correspondence between a TPM and a Platform.)

2045 c. Therefore, an EK is bound to a Platform. (i.e., there is a one to one correspondence  
2046 between an Endorsement Key and a Platform.)

## 2047 **11.3 Platform Identity and Privacy Considerations**

### 2048 **Start of informative comment**

2049The uniqueness property of cryptographic identities raises concerns that use of that identity  
2050could result in aggregation of activity logs. Analysis of the aggregated activity could reveal  
2051personal information that a user of a platform would not otherwise approve for distribution  
2052to the aggregators. Both EK and AIK identities have this property.

2053To counter undesired aggregation, TCG encourages the use of domain specific AIK keys and  
2054restricts the use of the EK key. The platform owner controls generation and distribution of  
2055AIK public keys.

2056If a digital signature was performed by the EK, then any entity could track the use of the  
2057EK. So use of the EK as a signature is cryptographically sound, but this does not ensure  
2058privacy. Therefore, a mechanism to allow verifiers (human or machine) to determine that  
2059the TPM really signed the message without using the EK is required.

### 2060 **End of informative comment**

## 2061 **11.4 Attestation Identity Keys**

### 2062 **Start of informative comment**

2063An Attestation Identity Key (AIK) is an alias for the Endorsement Key (EK). The EK cannot  
2064perform signatures for security reasons and due to privacy concerns.

2065 Generation of an AIK can occur anytime after establishment of the TPM Owner. The TPM  
2066 can create a virtually unlimited number of AIK.

2067 The TPM Owner controls all aspects of the generation and activation of an AIK. The TPM  
2068 Owner controls any data associated with the AIK. The AIK credential may contain  
2069 application specific information. The AIK must contain identification such that the TPM can  
2070 properly enforce the restrictions placed on an AIK.

2071 The AIK is an asymmetric key pair. For interoperability, the AIK is an RSA 2048-bit key. The  
2072 TPM must protect the private portion of the asymmetric key and ensure that the value is  
2073 never exposed. The user of an AIK must prove knowledge of the 160-bit AIK authorization  
2074 value to use the AIK.

2075 An AIK is a signature key, and is never used for encryption. It only signs information  
2076 generated internally by the TPM. The data could include PCR, other keys and TPM status  
2077 information. The AIK must never sign arbitrary external data, since it would be possible for  
2078 an attacker to create a block of data that appears to be a PCR value.

2079 AIK creation involves two TPM commands.

2080 The TPM\_MakeIdentity command causes the TPM to generate the AIK key pair. The  
2081 command also discloses the EK-AIK binding to the service that will issue the AIK credential.

2082 The TPM\_ActivateIdentity command unwraps a session key that allows for the decryption of  
2083 the AIK credential. The session key was encrypted using the PUBEK and requires the  
2084 PRIVEK to perform the decryption.

2085 Use of the AIK credential is outside of the control of the TPM.

2086 **End of informative comment**

2087 1. The TPM MUST permanently mark an AIK such that, for all subsequent uses of the AIK,  
2088 the AIK restrictions are enforced.

2089 2. An AIK MUST be:

- 2090 a. Statistically unique
- 2091 b. Difficult to forge or counterfeit
- 2092 c. Verifiable to challengers

2093 3. For interoperability the AIK MUST be

- 2094 a. An RSA 2048-bit key

2095 4. The AIK MUST only sign data generated by the TPM

## 2096 11.4.1 AIK Creation

2097 **Start of informative comment**

2098 As the AIK is an alias for the EK. The AIK creation process requires TPM Owner  
2099 authorization. The process actually requires two TPM Owner authorizations; creation and  
2100 credential activation.

2101 The AIK credential creation process is outside the control of the TPM. However, the  
2102 certification authority (CA) will attest (with the AIK credential) that the AIK is tied to valid  
2103 Endorsement, Platform and Conformance credentials.



326

2104Without these credentials, the AIK cannot prove that PCR values belong to a TPM. An owner  
2105may decide to trust any key generated by TPM\_MakeIdentity without activating the identity  
2106(e.g., because he is an administrator in a controlled company environment). In this case,  
2107the owner needs no credential. Another challenger can only trust that the AIK belongs to a  
2108TPM by seeing the credential of a trustworthy CA.

2109**End of informative comment**

21101. The TPM Owner MUST authorize the AIK creation process.

21112. The TPM MUST use a protected function to perform the AIK creation.

21123. The TPM Owner MUST indicate the entity that will provide the AIK credential as part of  
2113 the AIK creation process.

21144. The TPM Owner MAY indicate that NO credential will ever be created. If the TPM Owner  
2115 does indicate that no credential will be provided the TPM MUST ensure that no  
2116 credential can be created.

21175. The TTP MAY apply policies to determine if the presented AIK should be granted a  
2118 credential.

21196. The credential request package MUST be useable by only the Privacy CA selected by the  
2120 TPM Owner.

21217. The AIK credential MUST be only obtainable by the TPM that created the AIK credential  
2122 request.

2123 **11.4.2 AIK Storage**

2124**Start of informative comment**

2125The AIK may be stored on some general-purpose storage device.

2126When held outside of the TPM the AIK sensitive data must be encrypted and integrity  
2127protected.

2128**End of informative comment**

21291. When held outside of the TPM AIK encryption and integrity protection MUST protect the  
2130 AIK sensitive information

21312. The migration of AIK from one TPM to another MUST be prohibited

## 213212. Root of Trust for Storage (RTS)

### 2133Start of informative comment

2134The RTS provides protection on data in use by the TPM but held in external storage devices.  
2135The RTS provides confidentiality and integrity for the external blobs.

2136The RTS also provides the mechanism to ensure that the release of information only occurs  
2137in a named environment. The naming of an environment uses the PCR selection to  
2138enumerate the values.

2139Data protected by the RTS can migrate to other TPM.

### 2140End of informative comment

21411. The number and size of values held by the RTS SHOULD be limited only by the volume  
2142 of storage available on the platform

21432. The TPM MUST ensure that TPM\_PERMANENT\_DATA -> tpmProof is only inserted into  
2144 TPM internally generated and non-migratable information.

## 2145 12.1 Loading and Unloading Blobs

### 2146Start of informative comment

2147The TPM provides several commands to store and load RTS controlled data.

	Class	Command	Analog	Comment
1	Data / Internal / TPM	TPM_MakeIdentity	TPM_ActivateIdentity	Special purpose data
2	Data / External / TPM	TSS_Bind	TPM_Unbind	
3	Data / Internal / PCR	TPM_Seal	TPM_Unseal	
4	Data / External / PCR			
5	Key / Internal / TPM	TPM_CreateWrapKey	TPM_LoadKey	
6	Key / External / TPM	TSS_WrapKey	TPM_LoadKey	
7	Key / Internal / PCR			
8	Key / External / PCR	TSS_WrapKeyToPcr	TPM_LoadKey	

## 2148**13. Transport Sessions and Authorization Protocols**

### 2149**Start of informative comment**

2150The purpose of the authorization protocols and mechanisms is to prove to the TPM that the  
2151requestor has permission to perform a function and use some object. The proof comes from  
2152the knowledge of a shared secret.

2153AuthData is available for the TPM Owner and each entity (keys, for example) that the TPM  
2154controls. The AuthData for the TPM Owner and the SRK are held within the TPM itself and  
2155the AuthData for other entities are held with the entity.

2156The TPM Owner AuthData allows the Owner to prove ownership of the TPM. Proving  
2157ownership of the TPM does not immediately allow all operations – the TPM Owner is not a  
2158“super user” and additional AuthData must be provided for each entity or operation that  
2159has protection.

2160The TPM treats knowledge of the AuthData as complete proof of ownership of the entity. No  
2161other checks are necessary. The requestor (any entity that wishes to execute a command on  
2162the TPM or use a specific entity) may have additional protections and requirements where  
2163he or she (or it) saves the AuthData; however, the TPM places no additional requirements.

2164There are three protocols to securely pass a proof of knowledge of AuthData from requestor  
2165to TPM; the “Object-Independent Authorization Protocol” (OIAP), the “Object-Specific  
2166Authorization Protocol” (OSAP) and the “Delegate-Specific Authorization Protocol” (DSAP).  
2167The OIAP supports multiple authorization sessions for arbitrary entities. The OSAP  
2168supports an authentication session for a single entity and enables the confidential  
2169transmission of new authorization information. The DSAP supports the delegation of owner  
2170or entity authorization.

2171New authorization information is inserted by the “AuthData Insertion Protocol” (ADIP)  
2172during the creation of an entity. The “AuthData Change Protocol” (ADCP) and the  
2173“Asymmetric Authorization Change Protocol” (AACCP) allow the changing of the AuthData for  
2174an entity. The protocol definitions allow expansion of protocol types to additional TCG  
2175required protocols and vendor specific protocols.

2176The protocols use a “rolling nonce” paradigm. This requires that a nonce from one side be in  
2177use only for a message and its reply. For instance, the TPM would create a nonce and send  
2178that on a reply. The requestor would receive that nonce and then include it in the next  
2179request. The TPM would validate that the correct nonce was in the request and then create  
2180a new nonce for the reply. This mechanism is in place to prevent replay attacks and man-  
2181in-the-middle attacks.

2182The basic protocols do not provide long-term protection of AuthData that is the hash of a  
2183password or other low-entropy entities. The TPM designer and application writer must  
2184supply additional protocols if protection of these types of data is necessary.

2185The design criterion of the protocols is to allow for ownership authentication, command and  
2186parameter authentication and prevent replay and man-in-the-middle attacks.

2187The passing of the AuthData, nonces and other parameters must follow specific guidelines  
2188so that commands coming from different computer architectures will interoperate properly.

### 2189**End of informative comment**

21901. AuthData MUST use one of the following protocols
- 2191 a. OIAP
  - 2192 b. OSAP
  - 2193 c. DSAP
21942. Entity creation MUST use one of the following protocols
- 2195 a. ADIP
21963. Changing AuthData MUST use one of the following protocols
- 2197 a. ADCP
  - 2198 b. AACP
21994. The TPM MAY support additional protocols to authenticate, insert and change  
2200 AuthData.
22015. When a command has more than one AuthData value
- 2202 a. Each AuthData MUST use the same SHA-1 of the parameters
22036. Keys MAY specify authDataUsage -> TPM\_AUTH\_NEVER
- 2204 a. If the caller changes the tag from TPM\_TAG\_RQU\_AUTH1\_xxx to  
2205 TPM\_TAG\_RQU\_XXX the TPM SHALL ignore the AuthData values
  - 2206 b. If the caller leaves the tag as TPM\_TAG\_RQU\_AUTH1
    - 2207 i. The TPM will compute the AuthData based on the value store in the AuthData  
2208 location within the key, IGNORING the state of the AuthDataUsage flag.
    - 2209 c. Users may choose to use a well-known value for the AuthData when setting  
2210 AuthDataUsage to TPM\_AUTH\_NEVER.
    - 2211 d. If a key has AuthDataUsage set to TPM\_AUTH\_ALWAYS but is received in a  
2212 command with the tag TPM\_TAG\_RQU\_COMMAND, the command MUST return an  
2213 error code.
22147. For commands that normally have 2 authorization sessions, if the tag specifies only one  
2215 in the parameter array, then the first session listed is ignored (authDataUsage must be  
2216 TPM\_AUTH\_NEVER for this key) and the incoming session data is used for the second  
2217 auth session in the list.
22188. Keys MAY specify AuthDataUsage -> TPM\_NO\_READ\_PUBKEY\_AUTH
- 2219 a. If the key used in a command to read the public portion of the key (e.g.  
2220 TPM\_CertifyKey, TPM\_GetPubKey)
    - 2221 i. If the caller changes the tag from TPM\_TAG\_RQU\_AUTH1\_xxx to  
2222 TPM\_TAG\_RQU\_XXX, the TPM SHALL ignore the AuthData values
    - 2223 ii. If the caller leaves the tag as TPM\_TAG\_RQU\_AUTH1, the TPM will compute the  
2224 AuthData based on the value store in the AuthData location within the key,  
2225 IGNORING the state of the AuthDataUsage flag
  - 2226 b. else if the key used in command to read/access the private portion of the key(e.g.  
2227 TPM\_Sign)

344

- 2228           i. If the tag is TPM\_TAG\_RQU\_COMMAND, the command MUST return an error  
2229           code.

2230

## 2231 13.1 Authorization Session Setup

### 2232 **Start of informative comment**

2233 The TPM provides two protocols for authorizing the use of entities without revealing the  
2234 AuthData on the network or the connection to the TPM. In both cases, the protocol  
2235 exchanges nonce-data so that both sides of the transaction can compute a hash using  
2236 shared secrets and nonce-data. Each side generates the hash value and can compare to the  
2237 value transmitted. Network listeners cannot directly infer the AuthData from the hashed  
2238 objects sent over the network.

2239 The first protocol is the Object-Independent Authorization Protocol (OIAP), which allows the  
2240 exchange of nonces with a specific TPM. Once an OIAP session is established, its nonces  
2241 can be used to authorize the use of any entity managed by the TPM. The session can live  
2242 indefinitely until either party requests the session termination. The TPM\_OIAP function  
2243 starts the OIAP session.

2244 The second protocol is the Object Specific Authorization Protocol (OSAP). The OSAP allows  
2245 establishment of an authentication session for a single entity. The session creates nonces  
2246 that can authorize multiple commands without additional session-establishment overhead,  
2247 but is bound to a specific entity. The TPM\_OSAP command starts the OSAP session. The  
2248 TPM\_OSAP specifies the entity to which the authorization is bound.

2249 Most commands allow either form of authorization protocol. In general, however, the OIAP  
2250 is preferred – it is more generally useful because it allows usage of the same session to  
2251 provide authorization for different entities. The OSAP is, however, necessary for operations  
2252 that set or reset AuthData.

2253 OIAP sessions were designed for reasons of efficiency; only one setup process is required for  
2254 potentially many authorizations.

2255 An OSAP session is doubly efficient because only one setup process is required for  
2256 potentially many authorization calculations and the entity AuthData secret is required only  
2257 once. This minimizes exposure of the AuthData secret and can minimize human interaction  
2258 in the case where a person supplies the AuthData information. The disadvantage of the  
2259 OSAP is that a distinct session needs to be setup for each entity that requires authorization.  
2260 The OSAP creates an ephemeral secret that is used throughout the session instead of the  
2261 entity AuthData secret. The ephemeral secret can be used to provide confidentiality for the  
2262 introduction of new AuthData during the creation of new entities. Termination of the OSAP  
2263 occurs in two ways. Either side can request session termination (as usual) but the TPM  
2264 forces the termination of an OSAP session after use of the ephemeral secret for the  
2265 introduction of new AuthData.

2266 For both the OSAP and the OIAP, session setup is independent of the commands that are  
2267 authorized. In the case of OIAP, the requestor sends the TPM\_OIAP command, and with the  
2268 response generated by the TPM, can immediately begin authorizing object actions. The  
2269 OSAP is very similar, and starts with the requestor sending a TPM\_OSAP operation, naming  
2270 the entity to which the authorization session should be bound.

2271 The DSAP session is to provide delegated authorization information.

2272 All session types use a “rolling nonce” paradigm. This means that the TPM creates a new  
2273 nonce value each time the TPM receives a command using the session.

2274Example OIAP and OSAP sessions are used to illustrate session setup and use. The  
 2275fictitious command named TPM\_Example occupies the place where an ordinary TPM  
 2276command might be used, but does not have command specific parameters. The session  
 2277connects to a key object within the TPM. The key contains AuthData that will be used to  
 2278secure the session.

2279There could be as many as 2 authorization sessions applied to the execution of a single TPM  
 2280command or as few as 0. The number of sessions used is determined by TCG 1.2 Command  
 2281Specification and is indicated by the command ordinal parameter.

2282It is also possible to secure authorization sessions using ephemeral shared-secrets. Rather  
 2283than using AuthData contained in the stored object (e.g. key), the AuthData is supplied as a  
 2284parameter to OSAP session creation. In the examples below the key.usageAuth parameter is  
 2285replaced by the ephemeral secret.

2286**End of informative comment**

## 2287 13.2 Parameter Declarations for OIAP and OSAP Examples

2288**Start of informative comment**

2289To follow OIAP and OSAP protocol examples (Table 13:d and ), the reader should become  
 2290familiar with the parameters declared in Table 13:b and Table 13:c.

2291Several conventions are used in the parameter tables that may facilitate readability.

2292The Param column (Table 13:b) identifies the sequence in which parameters are packaged  
 2293into a command or response message as well as the size in bytes of the parameter value. If  
 2294this entry in the row is blank, that parameter is not included in the message. <> in the size  
 2295column means that the size of the element is variable. It is defined either explicitly by the  
 2296preceding parameter, or implicitly by the parameter type.

2297The HMAC column similarly identifies the parameters that are included in HMAC  
 2298calculations. This column also indicates the default parameters that are included in the  
 2299audit log. Exceptions are noted under the specific ordinal, e.g. TPM\_ExecuteTransport.

2300The HMAC # column details the parameters used in the HMAC calculation. Parameters 1S,  
 23012S, etc. are concatenated and hashed to inParamDigest or outParamDigest, implicitly called  
 23021H1 and possibly 1H2 if there are two authorization sessions. For the first session, 1H1,  
 23032H1, 3H1, and 4H1 are concatenated and HMAC'ed. For the second session, 1H2, 2H2,  
 23043H2, and 4H2 are concatenated and HMAC'ed.

2305In general, key handles are not included in HMAC calculations. This allows a lower  
 2306software layer to map the physical handle value generated by the TPM to a logical value  
 2307used by an upper software layer. The upper layer generally holds the HMAC key and  
 2308generates the HMAC. Excluding the key handle allows the mapping to occur without  
 2309breaking the HMAC. It is important to use a different authorization secret for each key to  
 2310prevent a man-in-the-middle from altering the key handle.

2311The Type column identifies the TCG data type corresponding to the passed value. An  
 2312encapsulation of the parameter type is not part of the command message.

2313The Name column is a fictitious variable name that aids in following the examples and  
 2314descriptions.

2315The double-lined row separator distinguishes authorization session parameters from  
2316command parameters. In Table 13:b the TPM\_Example command has three parameters;  
2317keyHandle, inArgOne and inArgTwo. The tag, paramSize and ordinal parameters are  
2318message header values describing contents of a command message. The parameters below  
2319the double-lined row are OIAP / OSAP /DSAP or transport authorization session related. If  
2320a second authorization session were used, the table would show a second authorization  
2321section delineated by a second double-lined row. The authorization session parameters  
2322identify shared-secret values, session nonces, session digest and flags.

2323In this example, a single authorization session is used signaled by the  
2324TPM\_TAG\_RQU\_AUTH1\_COMMAND tag.

2325For an OIAP or transport session, the TPM\_AUTHDATA description column specifies the  
2326HMAC key.

2327For an OSAP or DSAP session, the HMAC key is the shared secret that was calculated  
2328during the session setup, not the key specified in the description. The key specified in the  
2329description was previously used in the shared secret calculation.



362

2330

Param		HMAC		Type	Name	Description
#	Sz	#	Sz			
1	2			TPM_TAG	tag	TPM_TAG_RQU_AUTH1_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Command ordinal, fixed value of TPM_Example
4	4			TPM_KEY_HANDLE	keyHandle	Handle of a loaded key.
5	1	2S	1	BOOL	inArgOne	The first input argument
6	20	3S	20	UINT32	inArgTwo	The second input argument.
7	4			TPM_AUTHHANDLE	authHandle	The authorization handle used for keyHandle authorization.
		2H1	20	TPM_NONCE	authLastNonceEven	Even nonce previously generated by TPM to cover inputs
8	20	3 H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
9	1	4 H1	1	BOOL	continueAuthSession	The continue use flag for the authorization handle
10	20			TPM_AUTHDATA	inAuth	The AuthData digest for inputs and keyHandle. HMAC key: key.usageAuth.

2331

**Table 13:b - Authorization Protocol Input Parameters**

2332

Param		HMAC		Type	Name	Description
#	Sz	#	Sz			
1	2			TPM_TAG	Tag	TPM_TAG_RSP_AUTH1_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation. See section 4.3.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal, fixed value of TPM_Example
4	4	3S	4	UINT32	outArgOne	Output argument
5	20	2 H1	20	TPM_NONCE	nonceEven	Even nonce newly generated by TPM to cover outputs
		3 H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
6	1	4 H1	1	BOOL	continueAuthSession	Continue use flag, TRUE if handle is still active
7	20			TPM_AUTHDATA	resAuth	The AuthData digest for the returned parameters. HMAC key: key.usageAuth.

2333

**Table 13:c - Authorization Protocol Output Parameters**

2334End of informative comment

## 2335 **13.2.1 Object-Independent Authorization Protocol (OIAP)**

### 2336 **Start of informative comment**

2337 The purpose of this section is to describe the authorization-related actions of a TPM when it  
2338 receives a command that has been authorized with the OIAP protocol. OIAP uses the  
2339 TPM\_OIAP command to create the authorization session.

2340 Many commands use OIAP authorization. The following description is therefore necessarily  
2341 abstract. A fictitious TPM command, TPM\_Example is used to represent ordinary TPM  
2342 commands.

2343 Assume that a TPM user wishes to send command TPM\_Example. This is an authorized  
2344 command that uses the key denoted by keyHandle. The user must know the AuthData for  
2345 keyHandle (key.usageAuth) as this is the entity that requires authorization and this secret  
2346 is used in the authorization calculation. Let us assume for this example that the caller of  
2347 TPM\_Example does not need to authorize the use of keyHandle for more than one  
2348 command. This use model points to the selection of the OIAP as the authorization protocol.

2349 For the TPM\_Example command, the inAuth parameter provides the authorization to  
2350 execute the command. The following table shows the commands executed, the parameters  
2351 created and the wire formats of all of the information.

2352 <inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne,  
2353 inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode,  
2354 ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order:  
2355 authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the  
2356 following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

2357 There are two even nonces used to execute TPM\_Example, the one generated as part of the  
2358 TPM\_OAIP command (labeled authLastNonceEven below) and the one generated with the  
2359 output arguments of TPM\_Example (labeled as nonceEven below).

Caller	On the wire	Dir	TPM
Send TPM_OIAP	TPM_OIAP	→	Create session Create authHandle Associate session and authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle
Save authHandle, authLastNonceEven	authHandle, authLastNonceEven	←	Returns
Generate nonceOdd Compute inAuth = HMAC (key.usageAuth, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo authHandle nonceOdd continueAuthSession inAuth	→	TPM retrieves key.usageAuth (key must have been previously loaded) Verify authHandle points to a valid session, mismatch returns TPM_E_INVALIDAUTH Retrieve authLastNonceEven from internal session storage HM = HMAC (key.usageAuth, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_E_INVALIDAUTH Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC( key.usageAuth, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC( key.usageAuth, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2361

2362 Suppose now that the TPM user wishes to send another command using the same session.  
2363 For the purposes of this example, we will assume that the same example command is used  
2364 (ordinal = TPM\_Example). However, a different key (newKey) with its own secret  
2365 (newKey.usageAuth) is to be operated on. To re-use the previous session, the  
2366 continueAuthSession output boolean must be TRUE.

2367 The previous example shows the command execution reusing an existing authorization  
2368 session. The parameters created and the wire formats of all of the information.

2369 In this case, authLastNonceEven is the nonceEven value returned by the TPM with the  
2370 output parameters from the first protocol example.

2371

Caller	On the wire	Dir	TPM
Generate nonceOdd Compute inAuth = HMAC (newKey.usageAuth, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo nonceOdd continueAuthSession inAuth	→	TPM retrieves newKey.usageAuth (newKey must have been previously loaded) Retrieve authLastNonceEven from internal session storage HM = HMAC (newKey.usageAuth, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_E_INVALIDAUTH Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(newKey.usageAuth, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC( newKey.usageAuth, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2372

2373The TPM user could then use the session for further authorization sessions. Suppose,  
2374however, that the TPM user no longer requires the authorization session. There are three  
2375possibilities in this case:

2376The user issues a TPM\_Terminate\_Handle command to the TPM (section 5.3).

2377The input argument continueAuthSession can be set to FALSE for the last command. In  
2378this case, the output continueAuthSession value will be FALSE.

2379In some cases, the TPM automatically terminates the authorization session regardless of the  
2380input value of continueAuthSession. In this case as well, the output continueAuthSession  
2381value will be FALSE.

2382When an authorization session is terminated for any reason, the TPM invalidates the  
2383session's handle and terminates the session's thread (releases all resources allocated to the  
2384session).

2385**End of informative comment**

## 2386OIAP Actions

23871. The TPM MUST verify that the authorization handle (H, say) referenced in the command  
2388 points to a valid session. If it does not, the TPM returns the error code  
2389 TPM\_INVALID\_AUTHHANDLE

23902. The TPM SHALL retrieve the latest version of the caller's nonce (nonceOdd) and  
2391 continueAuthSession flag from the input parameter list, and store it in internal TPM  
2392 memory with the authSession 'H'.

23933. The TPM SHALL retrieve the latest version of the TPM's nonce stored with the  
2394 authorization session H (authLastNonceEven) computed during the previously executed  
2395 command.
23964. The TPM MUST retrieve the secret AuthData (SecretE, say) of the target entity. The entity  
2397 and its secret must have been previously loaded into the TPM.
23985. The TPM SHALL perform a HMAC calculation using the entity secret data, ordinal, input  
2399 command parameters and authorization parameters according to previously specified  
2400 normative regarding HMAC calculation.
24016. The TPM SHALL compare HM to the AuthData value received in the input parameters. If  
2402 they are different, the TPM returns the error code TPM\_AUTHFAIL if the authorization  
2403 session is the first session of a command, or TPM\_AUTH2FAIL if the authorization  
2404 session is the second session of a command. Otherwise, the TPM executes the command  
2405 which (for this example) produces an output that requires authentication.
24067. The TPM SHALL generate a nonce (nonceEven).
24078. The TPM creates an HMAC digest to authenticate the return code, return values and  
2408 authorization parameters to the same entity secret according to previously specified  
2409 normative regarding HMAC calculation.
24109. The TPM returns the return code, output parameters, authorization parameters and  
2411 AuthData digest.
241210. If the output continueUse flag is FALSE, then the TPM SHALL terminate the session.  
2413 Future references to H will return an error.

## 2414 **13.2.2 Object-Specific Authorization Protocol (OSAP)**

### 2415 **Start of informative comment**

2416 This section describes the actions of a TPM when it receives a TPM command via OSAP  
2417 session. Many TPM commands may be sent to the TPM via an OSAP session. Therefore, the  
2418 following description is necessarily abstract.

2419 The OSAP session is initialized through the creation of an ephemeral secret which is used to  
2420 protect session traffic. Sessions are created using the TPM\_OSAP command. This section  
2421 illustrates OSAP using a fictitious command called TPM\_Example.

2422 Assume that a TPM user wishes to send the TPM\_Example command to the TPM. The  
2423 keyHandle signifies that an OSAP session is being used and has the value “Auth1”. The  
2424 user must know the AuthData for keyHandle (key.usageAuth) as this is the entity that  
2425 requires authorization and this secret is used in the authorization calculation.

2426 Let us assume that the sender needs to use this key multiple times but does not wish to  
2427 obtain the key secret more than once. This might be the case if the usage AuthData were  
2428 derived from a typed password. This use model points to the selection of the OSAP as the  
2429 authorization protocol.

2430 For the TPM\_Example command, the inAuth parameter provides the authorization to  
2431 execute the command. The following table shows the commands executed, the parameters  
2432 created and the wire formats of all of the information.

2433 <inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne,  
2434 inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode,  
2435 ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order:  
2436 authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the  
2437 following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

2438 In addition to the two even nonces generated by the TPM (authLastNonceEven and  
2439 nonceEven) that are used for TPM\_OIAP, there is a third, labeled nonceEvenOSAP that is  
2440 used to generate the shared secret. For every even nonce, there is also an odd nonce  
2441 generated by the system.

Caller	On the wire	Dir	TPM
Send TPM_OSAP	TPM_OSAP keyHandle nonceOddOSAP	→	Create session & authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle Save the ADIP encryption scheme with authHandle Generate nonceEvenOSAP Generate sharedSecret = HMAC(key.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save keyHandle, sharedSecret with authHandle
Save authHandle, authLastNonceEven Generate sharedSecret = HMAC(key.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save sharedSecret	authHandle, authLastNonceEven nonceEvenOSAP	←	Returns
Generate nonceOdd & save with authHandle. Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo authHandle nonceOdd continueAuthSession inAuth	→	Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example and create returnCode. If TPM_Example requires ADIP encryption, use the algorithm indicated when the OSAP session was set up. Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2443

## 2444Table 13:d - Example OSAP Session

2445Suppose now that the TPM user wishes to send another command using the same session  
2446to operate on the same key. For the purposes of this example, we will assume that the same  
2447ordinal is to be used (TPM\_Example). To re-use the previous session, the  
2448continueAuthSession output boolean must be TRUE.

2449The following table shows the command execution, the parameters created and the wire  
2450formats of all of the information.

2451In this case, authLastNonceEven is the nonceEven value returned by the TPM with the  
2452output parameters from the first execution of TPM\_Example.

2453

Caller	On the wire	Dir	TPM
Generate nonceOdd Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal keyHandle inArgOne inArgTwo nonceOdd continueAuthSession inAuth	→	Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

2454

## 2455Table 13:e - Example Re-used OSAP Session

2456The TPM user could then use the session for further authorization sessions or terminate it  
2457in the ways that have been described above in TPM\_OIAP. Note that termination of the  
2458OSAP session causes the TPM to destroy the shared secret.

2459**End of informative comment**2460**OSAP Actions**

24611. The TPM MUST have been able to retrieve the shared secret (Shared, say) of the target  
2462 entity when the authorization session was established with TPM\_OSAP. The entity and  
2463 its secret must have been previously loaded into the TPM.

24642. The TPM MUST verify that the authorization handle (H, say) referenced in the command  
2465 points to a valid session. If it does not, the TPM returns the error code  
2466 TPM\_INVALID\_AUTHHANDLE.

24673. The TPM MUST calculate the HMAC (HM1, say) of the command parameters according  
2468 to previously specified normative regarding HMAC calculation.

24694. The TPM SHALL compare HM1 to the AuthData value received in the command. If they  
2470 are different, the TPM returns the error code TPM\_AUTHFAIL if the authorization session  
2471 is the first session of a command, or TPM\_AUTH2FAIL if the authorization session is the  
2472 second session of a command., the TPM executes command C1 which produces an  
2473 output (O, say) that requires authentication and uses a particular return code (RC, say).

24745. The TPM SHALL generate the latest version of the even nonce (nonceEven).

24756. The TPM MUST calculate the HMAC (HM2) of the return parameters according to  
2476 previously specified normative regarding HMAC calculation.



24777. The TPM returns HM2 in the parameter list.

24788. The TPM SHALL retrieve the continue flag from the received command. If the flag is  
2479 FALSE, the TPM SHALL terminate the session and destroy the thread associated with  
2480 handle H.

24819. If the shared secret was used to provide confidentiality for data in the received  
2482 command, the TPM SHALL terminate the session and destroy the thread associated with  
2483 handle H.

248410. Each time that access to an entity (key) is authorized using OSAP, the TPM MUST  
2485 ensure that the OSAP shared secret is that derived from the entity using TPM\_OSAP.

### 2486 **13.3 Authorization Session Handles**

#### 2487 **Start of informative comment**

2488 The TPM generates authorization handles to allow for the tracking of information regarding  
2489 a specific authorization invocation.

2490 The TPM saves information specific to the authorization, such as the nonce values,  
2491 ephemeral secrets and type of authentication in use.

2492 The TPM may create any internal representation of the handle that is appropriate for the  
2493 TPM's design. The requestor always uses the handle in the authorization structure to  
2494 indicate authorization structure in use.

2495 The TPM must support a minimum of two concurrent authorization handles. The use of  
2496 these handles is to allow the Owner to have an authorization active in addition to an active  
2497 authorization for an entity.

2498 To ensure garbage collection and the proper removal of security information, the requestor  
2499 should terminate all handles. Termination of the handle uses the continue-use flag to  
2500 indicate to the TPM that the handle should be terminated.

2501 Termination of a handle instructs the TPM to perform garbage collection on all AuthData.  
2502 Garbage collection includes the deletion of the ephemeral secret.

#### 2503 **End of informative comment**

2504 1. The TPM **MUST** support authorization handles. See Section 23 Session pool.

2505 2. The TPM **MUST** support authorization handle termination. The termination includes  
2506 secure deletion of all authorization session information.

## 2507**13.4 Authorization-Data Insertion Protocol (ADIP)**

### 2508**Start of informative comment**

2509The ADIP allows for the creation of new entities and the secure insertion of the new entity  
2510AuthData. The transmission of the new AuthData uses encryption with the key based on  
2511the shared secret of an OSAP session.

2512The creation of AuthData is the responsibility of the entity owner. He or she may use  
2513whatever process he or she wishes. The transmission of the AuthData from the entity owner  
2514to the TPM requires confidentiality and integrity. Since these requirements are not always  
2515met (e.g., because the insertion of the AuthData occurs over a network) additional measures  
2516have to be taken. The ADIP protocol ensures confidentiality of the AuthData, while the  
2517OSAP session HMAC provides integrity.

2518When ADIP uses the parent key shared secret, care must be taken when that secret is a  
2519well known value. In that case, it may be appropriate to wrap the ADIP command in a  
2520transport session.

2521When the requestor is sending the AuthData to the TPM, the command requires the  
2522authorization of the entity parent. For example, to create a new TPM identity key and set its  
2523AuthData requires the AuthData of the TPM Owner. To create a new wrapped key requires  
2524the AuthData of the parent key.

2525The creation of a new entity requires the authorization of the entity owner. When the  
2526requestor starts the creation process, the creator must establish an OSAP session using the  
2527parent of the new entity.

2528For the mandatory XOR encryption algorithm, the creator builds an encryption key using a  
2529SHA-1 hash of the OSAP shared secret and a session nonce. The creator XOR encrypts the  
2530new AuthData using the encryption key as a one-time pad and sends this encrypted data  
2531along with the creation request to the TPM. The TPM decrypts the AuthData using the  
2532same OSAP shared secret and session nonce.

2533The XOR encryption algorithm is sufficient for almost all use models. There may be  
2534additional use models where a different encryption algorithm would be beneficial. The TPM  
2535may support AES as an additional encryption algorithm. The key and IV or counter use the  
2536OSAP shared secret and session nonces.

2537The creator believes that the OSAP creates a shared secret known only to the creator and  
2538the TPM. The TPM believes that the creator is the entity owner by their knowledge of the  
2539parent entity AuthData. The creator believes that the process completed correctly and that  
2540the AuthData is correct because the HMAC will only verify with the OSAP shared secret.

2541In the following example, we want to send the previously described command  
2542TPM\_EXAMPLE to create a new entity. In the example, we assume there is a third input  
2543parameter encAuth, and that one of the input parameters is named parentHandle to  
2544reference the parent for the new entity (e.g., the SRK and its children).

2545

Caller	On the wire	Dir	TPM
Send TPM_OSAP	TPM_OSAP parentHandle nonceOddOSAP	→	Create session & authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle Save the ADIP encryption scheme with authHandle Generate nonceEvenOSAP Generate sharedSecret = HMAC(parent.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save parentHandle, sharedSecret with authHandle
Save authHandle, authLastNonceEven Generate sharedSecret = HMAC(parent.usageAuth, nonceEvenOSAP, nonceOddOSAP) Save sharedSecret	authHandle, authLastNonceEven nonceEvenOSAP	←	Returns
Generate nonceOdd & save with authHandle. Compute input parameter newAuth = XOR( entityAuthData, SHA1(sharedSecret, authLastNonceEven)) Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)			
Send TPM_Example	tag paramSize ordinal parentHandle inArgOne inArgTwo encAuth authHandle nonceOdd continueAuthSession inAuth	→	Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example: decrypt encAuth to entityAuth, create entity and build returnCode. Use the ADIP encryption scheme indicated when the OSAP session was set up. Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters Terminate the authorization session associated with authHandle

2546

2547 Table 13:f - Example ADIP Session

2548 **End of informative comment**

416

25491. The TPM MUST enable ADIP by using the OSAP or DSAP
- 2550 a. When an ordinal Action indicates that OSAP is required for the ADIP protocol (e.g.,  
2551 TPM\_CreateWrapKey), DSAP shall satisfy that requirement.
  - 2552 b. The upper byte of the entity type indicates the encryption scheme.
  - 2553 c. The TPM internally stores the encryption scheme as part of the session and enforces  
2554 the encryption choice on the subsequent use of the session.
  - 2555 d. When TPM\_ENTITY\_TYPE is used for ordinals other than TPM\_OSAP or TPM\_DSAP  
2556 (i.e., for cases where there is no ADIP encryption action), the TPM\_ENTITY\_TYPE  
2557 upper byte MUST be 0x00.
25582. The TPM MUST destroy the session whenever a new entity AuthData is created.
25593. The TPM MUST encrypt the AuthData for the new entity.
- 2560 a. The TPM MUST support the XOR encryption scheme.
  - 2561 b. The TPM MAY support AES symmetric key encryption schemes.
    - 2562 i. If TPM\_PERMANENT\_FLAGS -> FIPS is TRUE
    - 2563 (1) All encrypted authorizations MUST use a symmetric key encryption scheme.
  - 2564 a. Encrypted AuthData values occur in the following commands
    - 2565 i. TPM\_CreateWrapKey
    - 2566 ii. TPM\_ChangeAuth
    - 2567 iii. TPM\_ChangeAuthOwner
    - 2568 iv. TPM\_Seal
    - 2569 v. TPM\_Sealx
    - 2570 vi. TPM\_MakeIdentity
    - 2571 vii. TPM\_CreateCounter
    - 2572 viii. TPM\_CMK\_CreateKey
    - 2573 ix. TPM\_NV\_DefineSpace
    - 2574 (1) This ordinal contains a special case where no encryption is used.
    - 2575 x. TPM\_Delegate\_CreateKeyDelegation
    - 2576 xi. TPM\_Delegate\_CreateOwnerDelegation
25774. If the entity type indicates XOR encryption for the AuthData secret
- 2578 a. Create X1 the SHA-1 of the concatenation of (authHandle -> sharedSecret ||  
2579 authLastNonceEven).
  - 2580 b. Create the decrypted AuthData the XOR of X1 and the encrypted AuthData.
  - 2581 c. If the command ordinal contains a second AuthData2 secret (e.g.  
2582 TPM\_CreateWrapKey)
    - 2583 i. Create X2 the SHA-1 of the concatenation of (authHandle -> sharedSecret ||  
2584 nonceOdd).
    - 2585 ii. Create the decrypted AuthData2 the XOR of X2 and the encrypted AuthData2.

25865. If the entity type indicates symmetric key encryption
- 2587 a. The key for the encryption algorithm is the first bytes of the OSAP shared secret.
- 2588 i. E.g., For AES128, the key is the first 16 bytes of the OSAP shared secret.
- 2589 ii. There is no support for AES keys greater than 128 bits.
- 2590 b. If the entity type indicates CTR mode
- 2591 i. The initial counter value for AuthData is the first bytes of authLastNonceEven.
- 2592 (1) E.g., For AES128, the initial counter value is the first 16 bytes of
- 2593 authLastNonceEven.
- 2594 ii. If the command ordinal contains a second AuthData2 secret (e.g.
- 2595 TPM\_CreateWrapKey)
- 2596 (1) The initial counter value for AuthData2 is the first bytes of nonceOdd.
- 2597 iii. Additional counter values as required are generated by incrementing the counter
- 2598 value as described in 31.1.3 TPM\_ES\_SYM\_CTR.
- 2599

2600 **Start of informative comment**

2601 The method of incrementing the counter value is different from that used by some standard  
2602 crypto libraries (e.g. openssl, Java JCE) that increment the entire counter value. TPM  
2603 users should be aware of this to avoid errors when the counter wraps.

2604 **End of informative comment**

2605

## 2606**13.5 AuthData Change Protocol (ADCP)**

### 2607**Start of informative comment**

2608All entities from the Owner to the SRK to individual keys and data blobs have AuthData.  
2609This data may need to change at some point in time after the entity creation. The ADCP  
2610allows the entity owner to change the AuthData. The entity owner of a wrapped key is the  
2611owner of the parent key.

2612A requirement is that the owner must remember the old AuthData. The only mechanism to  
2613change the AuthData when the entity owner forgets the current value is to delete the entity  
2614and then recreate it.

2615To protect the data from exposure to eavesdroppers or other attackers, the AuthData uses  
2616the same encryption mechanism in use during the ADIP.

2617Changing AuthData requires opening two authentication handles. The first handle  
2618authenticates the entity owner (or parent) and the right to load the entity. This first handle  
2619is an OSAP and supplies the data to encrypt the new AuthData according to the ADIP  
2620protocol. The second handle can be either an OIAP or an OSAP, it authorizes access to the  
2621entity for which the AuthData is to be changed.

2622The AuthData in use to generate the OSAP shared secret must be the AuthData of the  
2623parent of the entity to which the change will be made.

2624When changing the AuthData for the SRK, the first handle OSAP must be setup using the  
2625TPM Owner AuthData. This is because the SRK does not have a parent, per se.

2626If the SRKAuth data is known to userA and userB, userA can snoop on userB while userB  
2627is changing the AuthData for a child of the SRK, and deduce the child's newAuth.  
2628Therefore, if SRKAuth is a well known value, TPM\_ChangeAuthAsymStart and  
2629TPM\_ChangeAuthAsymFinish are preferred over TPM\_ChangeAuth when changing  
2630AuthData for children of the SRK.

2631This applies to all children of the SRK, including TPM identities.

### 2632**End of informative comment**

26331. Changing AuthData for the TPM SHALL require authorization of the current TPM Owner.

26342. Changing AuthData for the SRK SHALL require authorization of the TPM Owner.

26353. If SRKAuth is a well known value, TPM\_ChangeAuth SHOULD NOT be used to change  
2636 the AuthData value of a child of the SRK, including the TPM identities.

26374. All other entities SHALL require authorization of the parent entity.

## 2638 **13.6 Asymmetric Authorization Change Protocol (AACP)**

### 2639 **Start of informative comment**

2640 This is now deprecated. Use the normal change session inside of a transport session with  
2641 confidentiality.

2642 This asymmetric change protocol allows the entity owner to change entity authorization,  
2643 under the parent's execution authorization, to a value of which the parent has no  
2644 knowledge.

2645 In contrast, the TPM\_ChangeAuth command uses the parent entity AuthData to create the  
2646 shared secret that encrypts the new AuthData for an entity. This creates a situation where  
2647 the parent entity ALWAYS knows the AuthData for entities in the tree below the parent.  
2648 There may be instances where this knowledge is not a good policy.

2649 This asymmetric change process requires two commands and the use of an authorization  
2650 session.

### 2651 **End of informative comment**

2652 1. Changing AuthData for the SRK SHALL involve authorization by the TPM Owner.

2653 2. If SRKAuth is a well known value,

2654 a. TPM\_ChangeAuthAsymStart and TPM\_ChangeAuthAsymFinish SHOULD be used to  
2655 change the AuthData value of a child of the SRK, including the TPM identities.

2656 3. All other entities SHALL involve authorization of the parent entity.



## 2657**14. FIPS 140 Physical Protection**

### 2658**Start of informative comment**

2659The FIPS 140-2 program provides assurance that a cryptographic device performs properly.  
2660It is appropriate for TPM vendors to attempt to obtain FIPS 140-2 certification.

2661The TPM design should be such that the TPM vendor has the opportunity of obtaining FIPS  
2662140-2 certification.

### 2663**End of informative comment**

## 2664**14.1 TPM Profile for FIPS Certification**

### 2665**Start of informative comment**

2666The FIPS mode of the TPM does require some changes over the normal TPM. These changes  
2667are listed here such that there is a central point of determining the necessary FIPS changes.

### 2668**Key creation and use**

2669TPM\_LoadKey, TPM\_CMK\_CreateKey and TPM\_CreateWrapKey changed to disallow the  
2670creation or loading of TPM\_AUTH\_NEVER, legacy and keys less than 1024 bits.  
2671TPM\_MakeIdentity changed to disallow TPM\_AUTH\_NEVER.

### 2672**End of informative comment**

26731. Each TPM Protected Capability MUST be designed such that some profile of the  
2674 Capability is capable of obtaining FIPS 140-2 certification

## 2675 **15. Maintenance**

### 2676 **Start of informative comment**

2677 The maintenance feature is a vendor-specific feature, and its implementation is vendor-  
2678 specific. The implementation must, however, meet the minimum security requirements so  
2679 that implementations of the maintenance feature do not result in security weaknesses.

2680 There is no requirement that the maintenance feature is available, but if it is implemented,  
2681 then the requirements must be met.

2682 The maintenance feature described in the specification is an example only, and not the only  
2683 mechanism that a manufacturer could implement that meets these requirements.

2684 Maintenance is different from backup/migration, because maintenance provides for the  
2685 migration of both migratory and non-migratory data. Maintenance is an optional TPM  
2686 function, but if a TPM enables maintenance, the maintenance capabilities in this  
2687 specification are mandatory – no other migration capabilities shall be used. Maintenance  
2688 necessarily involves the manufacturer of a Subsystem.

2689 When maintaining computer systems, it is sometimes the case that a manufacturer or its  
2690 representative needs to replace a Subsystem containing a TPM. Some manufacturers  
2691 consider it a requirement that there be a means of doing this replacement without the loss  
2692 of the non-migrational keys held by the original TPM.

2693 The owner and users of TCG platforms need assurance that the data within protected  
2694 storage is adequately protected against interception by third parties or the manufacturer.

2695 This process MUST only be performed between two platforms of the same manufacturer and  
2696 model. If the maintenance feature is supported, this section defines the required functions  
2697 defined at a high level. The final function definitions and entire maintenance process is left  
2698 to the manufacturer to define within the constraints of these high level functions.

2699 Any maintenance process must have certain properties. Specifically, any migration to a  
2700 replacement Subsystem must require collaboration between the Owner of the existing  
2701 Subsystem and the manufacturer of the existing Subsystem. Further, the procedure must  
2702 have adequate safeguards to prevent a non-migrational key being transferred to multiple  
2703 Subsystems.

2704 The maintenance capabilities TPM\_CreateMaintenanceArchive and  
2705 TPM\_LoadMaintenanceArchive enable the transfer of all Protected Storage data from a  
2706 Subsystem containing a first TPM (TPM<sub>1</sub>) to a Subsystem containing a second TPM (TPM<sub>2</sub>):

2707 A manufacturer places a public key in non-volatile storage into its TPMs at manufacture  
2708 time.

2709 The Owner of TPM<sub>1</sub> uses TPM\_CreateMaintenanceArchive to create a maintenance archive  
2710 that enables the migration of all data held in Protected Storage by TPM<sub>1</sub>. The Owner of TPM<sub>1</sub>  
2711 must provide his or her authorization to the Subsystem. The TPM then creates the  
2712 TPM\_MIGRATE\_ASYMKEY structure and follows the process defined.

2713 The XOR process prevents the manufacturer from ever obtaining plaintext TPM<sub>1</sub> data.

2714 The additional random data provides a means to assure that a maintenance process cannot  
2715 subvert archive data and hide such subversion.

2716The random mask can be generated by two methods, either using the TPM RNG or MGF1 on  
2717the TPM Owners AuthData.

2718The manufacturer takes the maintenance blob, decrypts it with its private key, and satisfies  
2719itself that the data bundle represents data from that Subsystem manufactured by that  
2720manufacturer. Then the manufacturer checks the endorsement certificate of TPM<sub>2</sub> and  
2721verifies that it represents a platform to which data from TPM<sub>1</sub> may be moved.

2722The manufacturer dispatches two messages.

2723The first message is made available to CAs, and is a revocation of the TPM<sub>1</sub> endorsement  
2724certificate.

2725The second message is sent to the Owner of TPM<sub>2</sub>, which will communicate the SRK,  
2726tpmProof and the manufacturer's permission to install the maintenance blob only on TPM<sub>2</sub>

2727The Owner uses TPM\_LoadMaintenanceArchive to install the archive copy into TPM<sub>2</sub>, and  
2728overwrite the existing TPM<sub>2</sub>-SRK and TPM<sub>2</sub>-tpmProof in TPM<sub>2</sub>. TPM<sub>2</sub> overwrites TPM<sub>2</sub>-SRK  
2729with TPM<sub>1</sub>-SRK, and overwrites TPM<sub>2</sub>-tpmProof with TPM<sub>1</sub>-tpmProof.

2730Note that the command TPM\_KillMaintenanceFeature prevents the operation of  
2731TPM\_CreateMaintenanceArchive and TPM\_LoadMaintenanceArchive. This enables an Owner  
2732to block maintenance (and hence the migration of non-migratory data) either to or from a  
2733TPM.

2734It is required that a manufacturer takes steps that prevent further access of migrated data  
2735by TPM<sub>1</sub>. This may be achieved by deleting the existing Owner from TPM<sub>1</sub>, for example.

2736For the manufacturer to validate that the maintenance blob is coming from a valid TPM, the  
2737manufacturer can require that a TPM identity sign the maintenance blob. The identity  
2738would be from a CA under the control of the manufacturer and hence the manufacturer  
2739would be satisfied that the blob is from a valid TPM.

#### 2740**End of informative comment**

27411. The maintenance feature MUST ensure that the information can be on only one TPM at  
2742 a time. Maintenance MUST ensure that at no time the process will expose a shielded  
2743 location. Maintenance MUST require the active participation of the Owner.

27442. Any migration of non-migratory data protected by a Subsystem SHALL require the  
2745 cooperation of both the Owner of that non-migratory data and the manufacturer of that  
2746 Subsystem. That manufacturer SHALL NOT cooperate in a maintenance process unless  
2747 the manufacturer is satisfied that non-migratory data will exist in exactly one  
2748 Subsystem. A TPM SHALL NOT provide capabilities that support migration of non-  
2749 migratory data unless those capabilities are described in the TCG specification.

27503. The maintenance feature MUST move the following

27514. TPM\_KEY for SRK. The maintenance process will reset the SRK AuthData to match the  
2752 TPM Owners AuthData

27535. TPM\_PERMANENT\_DATA -> tpmProof

27546. TPM Owner's authorization

## 2755 **15.1 Field Upgrade**

#### 2756**Start of informative comment**

2757 A TPM, once in the field, may need to update the protected capabilities. This command,  
2758 which is optional, provides the mechanism to perform the update.

2759 The goal is that field upgrade should only affect protected capabilities and not shielded  
2760 location, so that a patch can be applied without loss of user data. It is understood that this  
2761 goal may not be achievable in all cases.

2762 **End of informative comment**

2763 The TPM SHOULD have provisions for upgrading the subsystem after shipment from the  
2764 manufacturer. If provided the mechanism MUST implement the following guidelines:

2765 1. The upgrade mechanisms in the TPM MUST not require the TPM to hold a global secret.  
2766 The definition of global secret is a secret value shared by more than one TPM.

2767 2. The TPM is not allowed to pre-store or use unique identifiers in the TPM for the purpose  
2768 of field upgrade. The TPM MUST NOT use the endorsement key for identification or  
2769 encryption in the upgrade process. The upgrade process MAY use a TPM Identity (AIK) to  
2770 deliver upgrade information to specific TPM devices.

2771 3. The upgrade process SHOULD only change protected capabilities. The upgrade process  
2772 SHOULD NOT change shielded locations.

2773 4. The upgrade process SHOULD only access data in shielded locations where this data is  
2774 necessary to validate the TPM Owner, validate the TPME and manipulate the blob

2775 5. The TPM MUST conform to the TCG specification, protection profiles and security targets  
2776 after the upgrade. The upgrade MAY NOT decrease the security values from the original  
2777 security target.

2778 6. The security target used to evaluate this TPM MUST include this command in the TOE.

## 2779**16. Proof of Locality**

### 2780**Start of informative comment**

2781When a platform is designed with a trusted process, the trusted process may wish to  
2782communicate with the TPM and indicate that the command is coming from the trusted  
2783process. The definition of a trusted process is a platform specific issue.

2784The commands that the trusted process sends to the TPM are the normal TPM commands  
2785with a modifier that indicates that the trusted process initiated the command. The TPM  
2786accepts the command as coming from the trusted process merely because the modifier is  
2787set. The TPM itself is not responsible for how the signal is asserted; only that it honors the  
2788assertions. The TPM cannot verify the validity of the modifier.

2789The definition of the modifier is a platform specific issue. Depending on the platform, the  
2790modifier could be a special bus cycle or additional input pins on the TPM. The assumption  
2791is that spoofing the modifier to the TPM requires more than just a simple hardware attack,  
2792but would require expertise and possibly special hardware. One example would be special  
2793cycles on the LPC bus that inform the TPM it is under the control of a process on the PC  
2794platform.

2795To allow for multiple mechanisms and for finer grained reporting, the TPM will include 4  
2796locality modifiers. These four modifiers allow the platform specific specification to properly  
2797indicate exactly what is occurring and for TPM's to properly respond to locality.

### 2798**End of informative comment**

27991. The TPM modifies the receipt of a command and indicates that the trusted process sent  
2800 the command when the TPM determines that the modifier is on. The modifier MUST only  
2801 affect the individual command just received and MUST NOT affect any other commands.  
2802 However, TPM\_ExecuteTransport MUST propagate the modifier to the wrapped  
2803 command.

28042. A TPM platform specific specification MAY indicate the presence of a maximum of 4 local  
2805 modifiers. The modifier indication uses the TPM\_MODIFIER\_INDICATOR data type.

28063. The received modifier MUST indicate a single level.

28074. The definition of the trusted source is in the platform specific specification.

28085. For ease in reading this specification the indication that the TPM has received any  
2809 modifier will be LOCAL\_MOD = TRUE.

2810**17. Monotonic Counter**

2811**Start of informative comment**

2812The monotonic counter provides an ever-increasing incremental value. The TPM must  
2813support at least 4 concurrent counters. Implementations inside the TPM may create 4  
2814unique counters or there may be one counter with pointers to keep track of the pointers  
2815current value. A naming convention to allow for unambiguous reference to the various  
2816components the following terms are in use:

2817Internal Base – This is the main counter. It is in use internally by the TPM and is not  
2818directly accessible by any outside process.

2819External Counter – A counter in use by external processes. This could be related to the  
2820main counter via pointers and difference values or it could be a totally unique value. The  
2821value of an external counter is not affected by any use, increment or deletion of any other  
2822external counter.

2823Max Value – The max count value of all counters (internal and external). So if there were 3  
2824external counters having values of 10, 15 and 201 and the internal base having a value of  
2825201 then Max Value is 201. In the same example if the internal base was 502 then Max  
2826Value would be 502.

2827The external counter must allow for 7 years of increments every 5 seconds without causing  
2828a hardware failure. The TPM may create a throttling mechanism that limits the ability to  
2829increment an external counter within a certain time range. The output of the counter is a  
283032-bit value.

2831To create an external counter requires TPM Owner authorization. To increment an external  
2832counter the command must pass authorization to use the counter.

2833External counters can be tagged with a short text string to facilitate counter administration.

2834Manufacturers are free to implement the monotonic counter using any mechanism.

2835To illustrate the counters and base the following example is in use. This mechanism uses  
2836two saving values (diff and start), however this is only an example and not meant to indicate  
2837any specific implementation.



2838

2839The internal base (IB) always moves forward and can never be reset. IB drives all external  
2840counters on the machine.

2841The purpose of the following example is to show the two external counters always moving  
2842forward independent of the other and how the IB moves forward also.

2843Starting condition is that IB is at 22 and no other external counters are active.

2844Start external counter A

2845     Increment IB (set new Max Value) IB = 23

2846     Assign start value of A to 23 (or Max Value)

2847     Assign difference of A to 23 (we always start at current value of IB)

2848     Assign a handle for A

2849Increment A 5 times

2850     IB is now 28

2851Request current A value

2852     Return  $28 = 28 \text{ (IB)} + 23 \text{ (difference)} - 23 \text{ (start value)}$

2853     Counter A has gone from the start of 23 to 28 incremented 5 times.

2854TPM\_Startup(ST\_CLEAR)

2855Start Counter B

2856     Save A difference  $28 = 23 \text{ (old difference)} + 28 \text{ (IB)} - 23 \text{ (start value)}$

2857     Increment IB (set new Max Value) IB = 29

2858     Set start value of B to 29 (or Max Value)

2859     Assign difference of B to 29

2860     Assign handle for B

2861Increment B 8 times

2862     IB is now 37

2863Request B value

2864     Return  $37 = 37 \text{ (IB)} + 29 \text{ (difference)} - 29 \text{ (start value)}$

2865TPM\_Startup(ST\_CLEAR)

2866Increment A

2867     Store B difference (37)

2868     Load A start value of 37

2869     Increment IB to 38

2870Return A value

2871     Return  $29 = 38 \text{ (IB)} + 28 \text{ (difference)} - 37 \text{ (start value)}$

2872

2873 Notice that A has gone from 28 to 29 which is correct, while B is at 37. Depending on the  
2874 order of increments A may pass B or it may always be less than B.

2875 **End of informative comment**

2876 1. The counter MUST be designed to not wear out in the first 7 years of operation. The  
2877 counter MUST be able to increment at least once every 5 seconds. The TPM, in response  
2878 to operations that would violate these counter requirements, MAY throttle the counter  
2879 usage (cause a delay in the use of the counter) or return an error.

2880 2. The TPM MUST support at least 4 concurrent counters.

2881 3. The establishment of a new counter MUST prevent the reuse of any previous counter  
2882 value. I.E. if the TPM has 3 counters and the max value of a current counter is at 36  
2883 then the establishment of a new counter would start at 37.

2884 4. After a successful TPM\_Startup(ST\_CLEAR) the first successful TPM\_IncrementCounter  
2885 sets the counter handle. Any attempt to issue TPM\_IncrementCounter with a different  
2886 handle MUST fail.

2887 5. TPM\_CreateCounter does NOT set the counter handle.



## 2888**18. Transport Protection**

### 2889**Start of informative comment**

2890The creation of sessions allows for the grouping of a set of commands into a session. The  
2891session provides a log of all commands and can provide confidentiality of the commands  
2892using the session.

2893Session establishment creates a shared secret and then uses the shared secret to authorize  
2894and protect commands sent to the TPM using the session. The shared secret is passed to  
2895the TPM using an asymmetric encryption key. For best security, the caller should certify  
2896that the key is never available outside the TPM.

2897After establishing the session, the caller uses the session to wrap a command to execute.  
2898The user of the transport session can wrap any command except for commands that would  
2899create nested transport sessions.

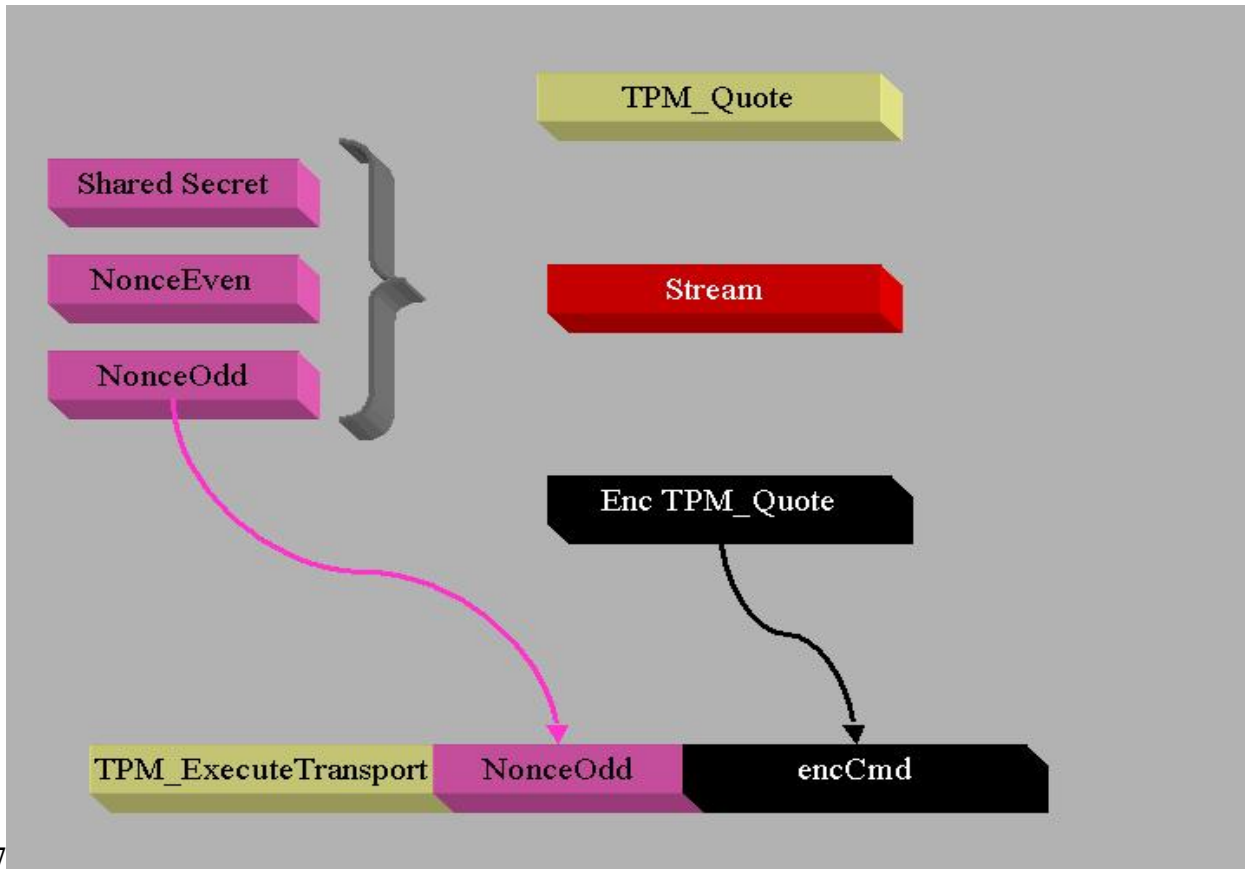
2900The log of executed commands uses a structure that includes the parameters and current  
2901tick count. The session log provides a record of each command using the session.

2902The transport session uses the same rolling nonce protocol that authorization sessions use.  
2903This protocol defines two nonces for each command sent to the TPM; nonceOdd provided by  
2904the caller and nonceEven generated by the TPM.

2905For confidentiality, the caller can use the MGF1 function to create an XOR string the same  
2906size as the command to execute. The inputs to the MGF1 function are the shared secret,  
2907nonceOdd and nonceEven. A symmetric key encryption algorithm can also be specified.

2908There is no explicit close session as the caller can use the continueSession flag set to false  
2909to end a session. The caller can also call the sign session log, which also ends the session. If  
2910the caller loses track of which sessions are active the caller should use the flush  
2911commands to regain control of the TPM resources.

2912For an attacker to successfully break the encryption the attacker must be able to determine  
2913from a few bits what an entire SHA-1 output was. This is equivalent to breaking SHA-1. The  
2914reason that the attacker will know some bits is that the commands are in a known format.  
2915This then allows the attacker to determine what the XOR bits were. Knowledge of 159 bits of  
2916the XOR stream does not provide any greater than 50% probability of knowing the 160<sup>th</sup> bit.



2917  
2918 This picture shows the protection of a TPM\_Quote command. Previously executed was  
2919 session establishment. The nonces in use for the TPM\_Quote have no relationship with the  
2920 nonces that are in use for the TPM\_ExecuteTransport command.

2921 **End of informative comment**

- 2922 1. The TPM MUST support a minimum of one transport session.
- 2923 2. The TPM MUST NOT support the nesting of transport sessions. The definition of nesting  
2924 is attempting to execute a wrapped command that is a transport session command. So  
2925 for example when executing TPM\_ExecuteTransport the wrapped command MUST not be  
2926 TPM\_ExecuteTransport.
- 2927 3. The TPM MUST ensure that if transport logging is active that the inclusion of the tick  
2928 count in the session log does not provide information that would make a timing attack  
2929 on the operations using the session more successful.
- 2930 4. The transport session MAY be exclusive. Any command executed outside of the exclusive  
2931 transport session MUST cause the invalidation of the exclusive transport session.
- 2932 a. The TPM\_ExecuteTransport command specifying the exclusive transport session is  
2933 the only command that does not terminate the exclusive session.
- 2934 5. It MAY be ineffective to wrap TPM\_SaveState in a transport session. Since the TPM MAY  
2935 include transport sessions in the saved state, the saved state MAY be invalidated by the  
2936 wrapping TPM\_ExecuteTransport.

## 2937**18.1 Transport encryption and authorization**

### 2938**Start of informative comment**

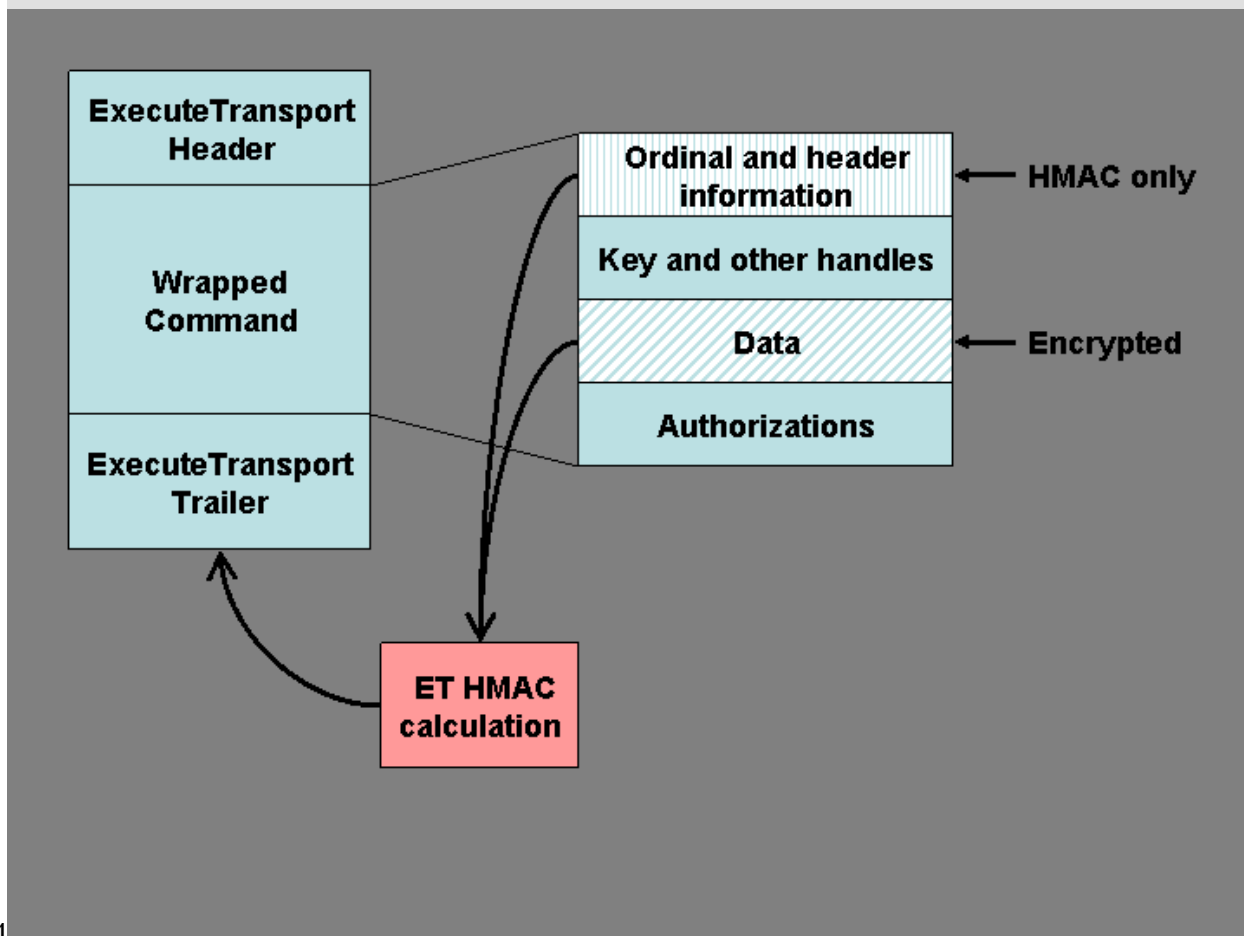
2939The confidentiality of the transport protection is provided by a encrypting the wrapped  
2940command. Encryption of various items in the wrapped command makes resource  
2941management of a TPM impossible. For this reason, encryption of the entire command is not  
2942possible. In addition to the encryption issue, there are difficulties with creating the HMAC  
2943for the TPM\_ExecuteTransport authorization.

2944The solution to these problems is to provide limited encryption and HMAC information.

2945The HMAC will only include two areas from the wrapped command, the command header  
2946information up to the handles, and the data after the handles. The format of all TPM  
2947commands is such that all handles are in the data stream prior to the payload or data. After  
2948the data comes the authorization information. To enable resource management, the HMAC  
2949for TPM\_ExecuteTransport only includes the ordinal, header information and the data. The  
2950HMAC does not include handles and the authorization handles and nonces.

2951The exception is TPM\_OwnerReadInternalPub, which uses fixed value key handles that are  
2952included in the encryption and HMAC calculation.

2953



2954

2955A more exact representation of the execute transport command would be the following

2956 \*\*\*\*\*  
2957 \* TAGet | LENet | ORDet | wrappedCmdSize | wrappedCmd | AUTHet \*  
2958 \*\*\*\*\*  
2959

2960 wrappedCmd looks like

2961 \*\*\*\*\*  
2962 \* TAGw | LENw | ORDw | HANDLESw(o) | DATAw | AUTH1w (o) | AUTH2w (o) \*  
2963 \*\*\*\*\*

2964 A more exact representation of the execute transport response would be the following

2965 \*\*\*\*\*  
2966 \* TAGet | LENet | RCet | ... | wrappedRspSize | wrappedRsp | AUTHet \*  
2967 \*\*\*\*\*  
2968

2969 wrappedRsp looks like

2970 \*\*\*\*\*  
2971 \* TAGw | LENw | RCw | HANDLESw(o) | DATAw | AUTH1w (o) | AUTH2w (o) \*  
2972 \*\*\*\*\*  
2973

2974 The calculation for AUTHet takes as the data component of the HMAC calculation the  
2975 concatenation of ORDw and DATAw. A normal HMAC calculation would have taken the  
2976 entire wrappedCmd value but for the executeTransport calculation only the above two  
2977 values are active. This does require the executeTransport command to parse the  
2978 wrappedCmd to find the appropriate values.

2979 The data for the command HMAC calculation is the following:

2980  $H1 = \text{SHA-1}(\text{ORDw} || \text{DATAw})$

2981  $\text{inParamDigest} = \text{SHA-1}(\text{ORDet} || \text{wrappedCmdSize} || H1)$

2982  $\text{AUTHet} = \text{HMAC}(\text{inParamDigest} || \text{lastNonceEven(et)} || \text{nonceOdd(et)} || \text{continue(et)})$

2983 The data for the response HMAC calculation is the following:

2984  $H2 = \text{SHA-1}(\text{RCw} || \text{ORDw} || \text{DATAw})$

2985  $\text{outParamDigest} = \text{SHA-1}(\text{RCet} || \text{ORDet} || \text{currentTicks} || \text{locality} || \text{wrappedRspSize} ||$   
2986  $H1)$

2987  $\text{AUTHet} = \text{HMAC}(\text{outParamDigest} || \text{nonceEven(et)} || \text{nonceOdd(et)} || \text{continue(et)})$

2988 DATAw is the unencrypted data. wrappedCmdSize and wrappedRspSize are the actual size  
2989 of the DATAw area and not the size of H1 or H2.

2990 **End of informative comment**

2991 The TPM MUST release a transport session and all information related to the session when:

2992 1. TPM\_ReleaseTransportSigned is executed

2993 2. TPM\_ExecuteTransport is executed with continueTransSession set to FALSE

2994 3. Any failure of the integrity check during execution of TPM\_ExecuteTransport

2995 4. If the session has TPM\_TRANSPORT\_LOG set and the TPM tick session is interrupted for  
2996 any reason. This is due to the return of tick values without the nonces associated with  
2997 the session.

29985. The TPM executes some command that deactivates the TPM or removes the TPM Owner  
2999 or EK.

## 3000**18.1.1 MGF1 parameters**

### 3001**Start of informative comment**

3002MGF1 provides the confidentiality for the transport session. MGF1 is a function from PKCS  
30031 version 2.0. This function provides a mechanism to distribute entropy over a large  
3004sequence. The sequence provides a value to XOR over the message. This in effect creates a  
3005stream cipher but not one that is available for bulk encryption.

3006Transport confidentiality uses MGF1 as a stream cipher and obtains the entropy for each  
3007message from the following three parameters; nonceOdd, nonceEven and session authData.

3008It is imperative that the stream cipher not use the same XOR sequence at any time. The  
3009following illustrates how the sequence changes for each message (both input and output).

3010M1Input – N2, N1, sessionSecret)

3011M1Output – N4, N1, sessionSecret)

3012M2Input – N4, N3, sessionSecret)

3013M2Output – N6, N3, sessionSecret)

3014There is an issue with this sequence. If the caller does not change N1 to N3 between  
3015M1Output and M2Input then the same sequence will be generated. The TPM does not  
3016enforce the requirement to change this value so it is possible to leak information.

3017The fix for this is to add one more parameter, the direction. So the sequence is now this:

3018M1Input – N2, N1, “in”, sessionSecret)

3019M1Output – N4, N1, “out”, sessionSecret)

3020M2Input – N4, N3, “in”, sessionSecret)

3021M2Output – N6, N3, “out”, sessionSecret)

3022Where “in” indicates the in direction and “out” indicates the out direction.

3023Notice the calculation for M1Output uses “out” and M2Input uses “in”, so if the caller  
3024makes a mistake and does not change nonceOdd, the sequence will still be different.

3025nonceEven is under control of the TPM and is always changing, so there is no need to worry  
3026about nonceEven not changing.

### 3027**End of informative comment**

## 3028**18.1.2 HMAC calculation**

### 3029**Start of informative comment**

3030The HMAC calculation for transports presents some issues with what should and should  
3031not be in the calculation. The idea is to create a calculation for the wrapped command and  
3032add that to the wrapper.

3033So the data area for a wrapped command is not entirely HMAC'd like a normal command  
3034would be.

3035 The process is to calculate the inParamDigest of the unencrypted wrapped command  
3036 according to the normal rules of command HMAC calculations. Then use that value as the  
3037 3S parameter in the calculation. 2S is the actual wrapped command size, and not the size  
3038 of inParamDigest.

3039 Example using a wrapped TPM\_LoadKey command

3040 Calculate the SHA-1 value for the TPM\_LoadKey command (ordinal and data) as per the  
3041 normal HMAC rules. Take the digest and use that value as 3S for the  
3042 TPM\_ExecuteTransport HMAC calculation.

3043 **End of informative comment**

### 3044 **18.1.3 Transport log creation**

3045 **Start of informative comment**

3046 The log of information that a transport session creates needs a mechanism to tie any keys  
3047 in use during the session to the session. As the HMAC and encryption for the command  
3048 specifically exclude handles, there is no direct way to create the binding.

3049 When creating the transport input log, if the handle(s) points to a key or keys, the public  
3050 keys are digested into the log. The session owner knows the value of any keys in use and  
3051 hence can still create a log that shows the values used by the log and can validate the  
3052 session.

3053 **End of informative comment**

### 3054 **18.1.4 Additional Encryption Mechanisms**

3055 **Start of informative comment**

3056 The TPM can optionally implement alternate algorithms for the encryption of commands  
3057 sent to the TPM\_ExecuteTransport command. The designation of the algorithm uses the  
3058 TPM\_ALGORITHM\_ID and TPM\_ENC\_SCHEME elements of the TPM\_TRANSPORT\_PUBLIC  
3059 parameter of the TPM\_EstablishTransport command.

3060 The anticipation is that AES will be supported by various TPM's. Symmetric algorithms  
3061 have options available to them like key size, block size and operating mode. When using an  
3062 algorithm other than MGF1 the algorithm and scheme must specify these options.

3063 **End of informative comment**

3064 1. The TPM MAY support other symmetric algorithms for the confidentiality requirement in  
3065 TPM\_EstablishTransport

## 3066 **18.2 Transport Error Handling**

3067 **Start of informative comment**

3068 With the transport hiding the actual execution of commands and the transport capable of  
3069 generating errors, rules must be established to allow for the errors and the results of  
3070 commands to be properly passed to TPM callers.

3071 **End of informative comment**

3072 1. There are 3 error cases:

30732. C1 is the case where an error occurs during the processing of the transport package at  
 3074 the TPM. In this case, the wrapped command has not been sent to the command  
 3075 decoder. Errors occurring during C1 are sent back to the caller as a response to the  
 3076 TPM\_ExecuteTransport command. The error response does not have confidentiality.

30773. C2 is the case where an error occurs during the processing of the wrapped command.  
 3078 This results in an error response from the command. The session returns the error  
 3079 response according to the attributes of the session.

30804. C3 is the case where an error occurs after the wrapped command has completed  
 3081 processing and the TPM is preparing the response to the TPM\_ExecuteTransport  
 3082 command. In this case, where the TPM does have an internal error, the TPM has no  
 3083 choice but to return the error as in C1. This however hides the results of the wrapped  
 3084 command. If the wrapped command completed successfully then there are session  
 3085 nonces that are being returned to the caller that are lost. The loss of these nonces  
 3086 causes the caller to be unsure of the state of the TPM and requires the reestablishment  
 3087 of sessions and keys.

### 3088**18.3 Exclusive Transport Sessions**

#### 3089**Start of informative comment**

3090The caller may establish an exclusive session with the TPM. When an exclusive session is  
 3091running, execution of any command other than TPM\_ExecuteTransport or  
 3092TPM\_ReleaseTransportSigned targeting the exclusive session causes the abnormal  
 3093invalidation of the exclusive transport session. Invalidation means that the handle is no  
 3094longer valid and all subsequent attempts to use the handle return an error.

3095The design for the exclusive session provides an assurance that no other command  
 3096executed on the TPM. It is not a lock to prevent other operations from occurring. Therefore,  
 3097the caller is responsible for ensuring no interruption of the sequence of commands using  
 3098the TPM.

#### 3099**One exclusive session**

3100The TPM only supports one exclusive session at a time. There is no nesting or other  
 3101commands possible. The TPM maintains an internal flag that indicates the existence of an  
 3102exclusive session.

#### 3103**TSS responsibilities**

3104It is the responsibility of the TSS (or other controlling software) to ensure that only  
 3105commands using the session reach the TPM. As the purpose of the session is to show that  
 3106nothing else occurred on the TPM during the session, the TSS should control access to the  
 3107TPM and prevent any other uses of the TPM. The TSS design must take into account the  
 3108possibility of exclusive session handle invalidation.

#### 3109**Sleep states**

3110Exclusive sessions as defined here do not work across TPM\_SaveState and  
 3111TPM\_Startup(ST\_STATE) invocations. To have this sequence work properly there would  
 3112need to be exceptions to allowing only TPM\_ExecuteTransport and  
 3113TPM\_ReleaseTransportSigned in an exclusive session. The requirement for these exceptions  
 3114would come from the attempt of the TSS to understand the current state of the TPM.  
 3115Commands like TPM\_GetCapability and others would have to execute to inform the TSS as

3116 to the internal state of the TPM. For this reason, there are no exceptions to the rule and the  
3117 exclusive session does not remain active across a TPM\_SaveState command.

3118 **End of informative comment**

3119 1. The TPM MUST support only one exclusive transport session

3120 2. The TPM MUST invalidate the exclusive transport session upon the receipt of any  
3121 command other than TPM\_ExecuteTransport or TPM\_ReleaseTransportSigned targeting  
3122 the exclusive session.

3123 a. Invalidation includes the release of any resources assigned to the session

3124 **18.4 Transport Audit Handling**

3125 **Start of informative comment**

3126 Auditing of TPM\_ExecuteTransport occurs as any other command that may require  
3127 auditing. There are two entries in the log, one for input one for output. The execution of the  
3128 wrapped command can create an anomaly in the log.

3129 Assume that both TPM\_ExecuteTransport and the wrapped commands require auditing, the  
3130 audit flow would look like the following:

3131 TPM\_ExecuteTransport input parameters

3132 wrapped command input parameters

3133 wrapped command output parameters

3134 TPM\_ExecuteTransport output parameters

3135 **End of informative comment**

3136 1. Audit failures are reported using the AUTHFAIL error commands and reflect the success  
3137 or failure of the wrapped command.

3138 **18.4.1 Auditing of wrapped commands**

3139 **Start of informative comment**

3140 Auditing provides information to allow an auditor to recreate the operations performed.  
3141 Confidentiality on the transport channel is to hide what operations occur. These two  
3142 features are in conflict. According to the TPM design philosophy, the TPM Owner takes  
3143 precedence.

3144 For a command sent on a transport session, with the session using confidentiality and the  
3145 command requiring auditing, the TPM will execute the command however the input and  
3146 output parameters for the command are ignored.

3147 **End of informative comment**

3148 1. When the wrapped command requires auditing and the transport session specifies  
3149 encryption, the TPM MUST perform the audit. However, when computing the audit  
3150 digest:

3151 a. For input, only the ordinal is audited.

3152 b. For output, only the ordinal and return code are audited.



## 3153 **19. Audit Commands**

### 3154 **Start of informative comment**

3155To allow the TPM Owner the ability to determine that certain operations on the TPM have  
3156been executed, auditing of commands is possible. The audit value is a digest held internally  
3157to the TPM and externally as a log of all audited commands. With the log held externally to  
3158the TPM, the internal digest must allow the log auditor to determine the presence of attacks  
3159against the log. The evidence of tampering may not provide evidence of the type of attack  
3160mounted against the log.

3161The TPM cannot enforce any protections on the external log. It is the responsibility of the  
3162external log owner to properly maintain and protect the log.

3163The TPM provides mechanisms for the external log maintainer to resynchronize the internal  
3164digest and external logs.

3165The Owner has the ability to set which functions generate an audit event and to change  
3166which functions generate the event at any time.

3167The status of the audit generation is not sensitive information and so the command to  
3168determine the status of the audit generation is not an owner authorized command.

3169It is important to note the difference between auditing and the logging of transport sessions.  
3170The audit log provides information on the execution of specific commands. There will be a  
3171very limited number of audited commands, most likely those commands that provide  
3172identities and control of the TPM. Commands such as TPM\_Unseal would not be audited.  
3173They would use the logging functions of a transport session.

3174The auditing of an ordinal happens in a two-step process. The first step involves auditing  
3175the receipt of the command and the input parameters; the second step involves auditing the  
3176response to the command and the output parameters.

3177There is a requirement to enable verification of the external audit log both during a power  
3178session and across power sessions and to enable detection of partial or inconsistent audit  
3179logs throughout the lifetime of a TPM.

3180A TPM will hold an internal record consisting of a non-volatile counter (that increments  
3181once per session, when the first audit event of that session occurs) and a digest (that holds  
3182the digest of the current session). Most probably, the audit digest will be volatile. Note,  
3183however, that nothing in this specification prevents the use of a non-volatile audit digest.  
3184This arrangement of counter and digest is advantageous because it is easier to build a high  
3185endurance non-volatile counter than a high endurance non-volatile digest. This  
3186arrangement is insufficient, however, because the truncation of an audit log of any session  
3187is possible without trace. It is therefore necessary to perform an explicit close on the audit  
3188session. If there is no record of a close-audit event in an audit session, anything could have  
3189happened after the last audit event in the audit log. The essence of a typical TPM audit  
3190recording mechanism is therefore:

3191The TPM contains a volatile digest used like a PCR, where the “integrity metrics” are digests  
3192of command parameters in the current audit session.

3193An audit session opens when the volatile “PCR” digest is “extended” from its NULL state.  
3194This occurs whenever an audited command is executed AND no audit session currently

exists, and in no other circumstances. When an audit session opens, a non-volatile counter is automatically incremented.

An audit session closes when a TPM receives TPM\_GetAuditDigestSigned with a closeAudit parameter asserted. An audit session must be considered closed if the value in the volatile digest is invalid (for whatever reason).

TPM\_GetCapability should report the effect of TPM\_Startup on the volatile digest. (TPMs may initialize the volatile digest on the first audit command after TPM\_Startup(ST\_CLEAR), or on the first audit command after any version of TPM\_Startup, or may be independent of TPM\_Startup.)

When the TPM signs its audit digest, it signs the concatenation of the non-volatile counter and the volatile digest, and exports the value of the non-volatile counter, plus the value of the volatile digest, plus the value of the signature.

If the audit digest is initialized by TPM\_Startup(ST\_STATE), then it may be useless to audit the TPM\_SaveState ordinal. Any command after TPM\_SaveState MAY invalidate the saved state. If authorization sessions are part of the saved state, TPM\_GetAuditDigestSigned will most likely invalidate the state as it changes the preserved authorization session nonce. It may therefore be impossible to get the audit results.

The system designer needs to ensure that the selected TPM can handle the specific environment and avoid burnout of the audit monotonic counter.

#### End of informative comment

1. Audit functionality is optional

a. If the platform specific specification requires auditing, the specification SHALL indicate how the TPM implements audit

2. The TPM MUST maintain an audit monotonic count that is only available for audit purposes.

a. The increment of this audit counter is under the sole control of the TPM and is not usable for other count purposes.

b. This monotonic count MUST BE incremented by one whenever the audit digest is “extended” from a NULL state.

3. The TPM MUST maintain an audit digest.

a. This digest MUST be set to all zeros upon the execution of TPM\_GetAuditDigestSigned with a TRUE value of closeAudit provided that the signing key is an identity key.

b. This digest MAY be set to all zeros on TPM\_Startup[ST\_CLEAR] or TPM\_Startup[ST\_STATE].

c. When an audited command is executed, this register MUST be extended with the digest of that command.

4. Each command ordinal has an indicator in non-volatile TPM memory that indicates if execution of the command will generate an audit event. The setting of the ordinal indicator MUST be under control of the TPM Owner.

## 3235**19.1 Audit Monotonic Counter**

### 3236**Start of informative comment**

3237The audit monotonic counter (AMC) performs the task of sequencing audit logs across audit  
3238sessions. The AMC must have no other uses other than the audit log.

3239The TPM and platform should be matched such that the expected AMC endurance matches  
3240the expected platform audit sessions and sleep cycles.

3241Given the size of the AMC it is not anticipated that the AMC would roll over. If the AMC  
3242were to roll over, and the storage of the AMC still allowed updates, the AMC could cycle and  
3243start at 0 again.

### 3244**End of informative comment**

32451. The AMC is a TPM\_COUNTER\_VALUE.

32462. The AMC MUST last for 7 years or at least 1,000,000 audit sessions, whichever occurs  
3247 first. After this amount of usage, there is no guarantee that the TPM will continue to  
3248 properly increment the monotonic counter.

## 3249 **20. Design Section on Time Stamping**

### 3250 **Start of informative comment**

3251 The TPM provides a service to apply a time stamp to various blobs. The time stamp provided  
3252 by the TPM is not an actual universal time clock (UTC) value but is the number of timer  
3253 ticks the TPM has counted. It is the responsibility of the caller to associate the ticks to an  
3254 actual UTC time.

3255 The TPM counts ticks from the start of a timing session. Timing sessions are platform  
3256 dependent events that may or may not coincide with TPM\_Init and TPM\_Startup sessions.  
3257 The reason for this difference is the availability of power to the TPM. In a PC desktop, for  
3258 instance power could be continually available to the TPM by using power from the wall  
3259 socket. For a PC mobile platform, power may not be available when only using the internal  
3260 battery. It is a platform designer's decision as to when and how they supply power to the  
3261 TPM to maintain the timing ticks.

3262 The TPM can provide a time stamping service. The TPM does not maintain an internal  
3263 secure source of time rather the TPM maintains a count of the number of ticks that have  
3264 occurred since the start of a timing session.

3265 On a PC, the TPM may use the timing source of the LPC bus or it may have a separate clock  
3266 circuit. The anticipation is that availability of the TPM timing ticks and the tick resolution is  
3267 an area of differentiation available to TPM manufactures and platform providers.

### 3268 **End of informative comment**

3269 1. This specification makes no requirement on the mechanism required to implement the  
3270 tick counter in the TPM.

3271 2. This specification makes no requirement on the ability for the TPM to maintain the  
3272 ability to increment the tick counter across power cycles or in different power modes on  
3273 a platform.

## 3274 **20.1 Tick Components**

### 3275 **Start of informative comment**

3276 The TPM maintains for each tick session the following values:

3277 Tick Count Value (TCV) – The count of ticks for the session.

3278 Tick Increment Rate (TIR) – The rate at which the TCV is incremented. There is a set  
3279 relationship between TIR and seconds, the relationship is set during manufacturing of the  
3280 TPM and platform. This is the TPM\_CURRENT\_TICKS -> tickRate parameter.

3281 Tick Session Nonce (TSN) – The session nonce is set at the start of each tick session.

### 3282 **End of informative comment**

3283 1. The TCV MUST be set to 0 at the start of each tick session. The TPM MUST start a new  
3284 tick session if the TPM loses the ability to increment the TCV according to the TIR.

3285 2. The TSN MUST be set to the next value from the TPM RNG at the start of each new tick  
3286 session. When the TPM loses the ability to increment the TCV according to the TIR the  
3287 TSN MUST be set to all zeros.

32883. If the TPM discovers tampering with the tick count (through timing changes etc) the TPM  
 3289 MUST treat this as an attack and shut down further TPM processing as if a self-test had  
 3290 failed.

## 3291**20.2 Basic Tick Stamp**

### 3292**Start of informative comment**

3293The TPM does not provide a secure time source, nor does it provide a signature over some  
 3294time value. The TPM does provide a signature over some current tick counter. The signature  
 3295covers a hash of the blob to stamp, the current counter value, the tick session nonce and  
 3296some fixed text.

3297The Tick Stamp Result (TSR) is the result of the tick stamp operation that associates the  
 3298TCV, TSN and the blob. There is no association with the TCV or TSR with any UTC value at  
 3299this point.

### 3300**End of informative comment**

## 3301**20.3 Associating a TCV with UTC**

### 3302**Start of informative comment**

3303An outside observer would like to associate a TCV with a relevant time value. The following  
 3304shows how to accomplish this task. This protocol is not required but shows how to  
 3305accomplish the job.

3306EntityA wants to have BlobA time stamped. EntityA performs TPM\_TickStamp on BlobA.  
 3307This creates TSRB (TickStampResult for Blob). TSRB records TSRBTCV, the current value of  
 3308the TCV, and associates TSRBTCV with the TSN.

3309Now EntityA needs to associate a TCV with a real time value. EntityA creates blob TS which  
 3310contains some known text like "Tick Stamp". EntityA performs TPM\_TickStamp on blob TS  
 3311creating TSR1. This records TSR1TCV, the current value of the TCV, and associates  
 3312TSR1TCV with the TSN.

3313EntityA sends TSR1 to a Time Authority (TA). TA creates TA1 which associates TSR1 with  
 3314UTC1.

3315EntityA now performs TPM\_TickStamp on TA1. This creates TSR2. TSR2 records TSR2TCV,  
 3316the current values of the TCV, and associates TSR2TCV with the TSN.

### 3317**Analyzing the associations**

3318EntityA has three TSR's; TSRB the TSR of the blob that we wanted to time stamp, TSR1 the  
 3319TSR associated with the TS blob and TSR2 the TSR associated with the information from  
 3320the TA. EntityA wants to show an association between the various TSR such that there is a  
 3321connection between the UTC and BlobA.

3322From TSR1 EntityA knows that TSR1TCV is less than the UTC. This is true since the TA is  
 3323signing TSR1 and the creation of TSR1 has to occur before the signature of TSR1. Stated  
 3324mathematically:

$$3325 \quad \text{TSR1TCV} < \text{UTC1}$$

3326From TSR2 EntityA knows that TSR2TCV is greater than the UTC. This is true since the  
 3327TPM is signing TA1 which must be created before it was signed. Stated mathematically:

3328  $TSR2TCV > UTC1$

3329 EntityA now knows  $TSR1TCV$  and  $TSR2TCV$  bound  $UTC1$ . Stated mathematically:

3330  $TSR1TCV < UTC1 < TSR2TCV$

3331 This association holds true if the TSN for  $TSR1$  matches the TSN for  $TSR2$ . If some event  
3332 occurs that causes the TPM to create a new TSN and restart the TCV then EntityA must  
3333 start the process all over again.

3334 EntityA does not know when  $UTC1$  occurred in the interval between  $TSR1TCV$  and  
3335  $TSR2TCV$ . In fact, the value  $TSR2TCV$  minus  $TSR1TCV$  ( $TSRDELTA$ ) is the amount of  
3336 uncertainty to which a TCV value should be associated with  $UTC1$ . Stated mathematically:

3337  $TSRDELTA = TSR2TCV - TSR1TCV$  iff  $TSR1TSN = TSR2TSN$

3338 EntityA can obtain  $k1$  the relationship between ticks and seconds using the  
3339 `TPM_GetCapability` command. EntityA also obtains  $k2$  the possible errors per tick. EntityA  
3340 now calculate  $DeltaTime$  which is the conversion of ticks to seconds and the  $TSRDELTA$ .  
3341 State mathematically:

3342  $DeltaTime = (k1 * TSRDELTA) + (k2 * TSRDELTA)$

3343

3344 To make the association between  $DeltaTime$ ,  $UTC$  and  $TSRB$  note the following:

3345  $DeltaTime = (k1 * TSRDelta) + Drift = TimeChange + Drift$

3346 Where  $ABSOLUTEVALUE(Drift) < k2 * TSRDelta$

3347 (1)  $TSR1TCV < UTC1 < TSR2TCV$

3348 True since you cannot sign something before it exists

3349 (2)  $TSR1TCV < UTC1 < TSR1TCV + TSR2TCV - TSR1TCV \leq TSR1TCV + DeltaTime (=$   
3350  $TSR1TCV + TimeChange + Drift)$

3351 True because  $TSR1$  and  $TSR2$  are in the same tick session proved by the same TSN. (Note  
3352  $TimeChange$  is positive!)

3353 (3)  $0 < UTC1 - TSR1TCV < DeltaTime$

3354 (Subtract  $TSR1TCV$  from all sides)

3355 (4)  $0 > TSR1TCV - UTC1 > -DeltaTime = -TimeChange - Drift$

3356 (Multiply through by -1)

3357 (5)  $TimeChange/2 > [TSR1TCV - (UTC1 - TimeChange/2)] > -TimeChange/2 - Drift$

3358 (add  $TimeChange/2$  to all sides)

3359 (6)  $TimeChange/2 + ABSOLUTEVALUE(Drift) > [TSR1TCV - (UTC1 - TimeChange/2)]$

3360  $> -TimeChange/2 - ABSOLUTEVALUE(Drift)$

3361 Making the large side of an equality bigger, and potentially making the small side smaller.

3362 (7)  $ABSOLUTEVALUE[TSR1TCV - (UTC1 - TimeChange/2)] < TimeChange/2 +$

3363  $ABSOLUTEVALUE(Drift)$

3364 (Definition of Absolute Value, and  $TimeChange$  is positive)

533  
3365  
3366From which we see that  $TSR1TCV$  is approximately  $UTC1 - TimeChange/2$  with a symmetric  
3367possible error of  $TimeChange/2 + AbsoluteValue(Drift)$   
3368We can calculate this error as being less than  $k1 * TSRDelta/2 + k2 * TSRDelta$ .  
3369  
3370EntityA now has the ability to associate  $UTC1$  with  $TSBTSV$  and by allow others to know  
3371that BlobA was signed at a certain time. First  $TSBTSN$  must equal  $TSR1TSN$ . This  
3372relationship allows EntityA to assert that  $TSRB$  occurs during the same session as  $TSR1$   
3373and  $TSR2$ .  
3374EntityA calculates  $HashTimeDelta$  which is the difference between  $TSR1TCV$  and  $TSRBTCV$   
3375and the conversion of ticks to seconds.  $HashTimeDelta$  includes the same  $k1$  and  $k2$  as  
3376calculated above. Stated mathematically:  
3377                     $E = k2(TSR1TCV - TSRBTCV)$   
3378                     $HashTimeDelta = k1(TSR1TCV - TSRBTCV) + E$   
3379Now the following relationships hold:  
3380(1)  $UTC1 - DeltaTime < TSRBTCV - (TSRBTCV - TSR1TCV) < UTC1$   
3381(2)  $UTC1 - DeltaTime < TSRBTCV + HashTimeDelta + E < UTC1$   
3382(3)  $UTC1 - HashTimeDelta - DeltaTime - E < TSRBTCV < UTC1 - HashTimeDelta + E$   
3383(4)  $TSRBTCV = (UTC1 - HashTimeDelta - DeltaTime/2) + (E + DeltaTime/2)$   
3384This has the correct properties  
3385As  $DeltaTime$  grows so does the error bar (or the uncertainty of the time association)  
3386As the difference between the time of the measurement and the time of the time stamp  
3387grows, so does the  $E$  as a function of  $E$  is  $HashTimeDelta$   
3388**End of informative comment**

## 3389**20.4 Additional Comments and Questions**

### 3390**Start of informative comment**

#### 3391**Time Difference**

3392If two things are time stamped, say at  $TCVs$  and  $TCVe$  (for  $TCV$  at start,  $TCV$  at end) then  
3393any entity can calculate the time difference between the two events and will get:

3394             $TimeDiff = k1 * |TCVe - TCVs| + k2 * |TCVe - TCVs|$

3395This  $TimeDiff$  does not indicate what time the two events occurred at it merely gives the  
3396time between the events. This time difference doesn't require a Time Authority.

#### 3397**Why is TSN (tick session nonce) required?**

3398Without it, there is no way to associate a Time Authority stamp with any  $TSV$ , as the  $TSV$   
3399resets at the start of every tick session. The  $TSN$  proves that the concatenation of  $TSV$  and  
3400 $TSN$  is unique.

#### 3401**How does the protocol prevent replay attacks?**

3402 The TPM signs the TSR sent to the TA. This TSR contains the unique combination of TSV  
3403 and TSN. Since the TSN is unique to a tick session and the TSV continues to increment any  
3404 attempt to recreate the same TSR will fail. If the TPM is reset such that the TSV is at the  
3405 same value, the TSN will be a new value. If the TPM is not reset then the TSV continues to  
3406 increment and will not repeat.

#### 3407 **How does EntityA know that the TSR1 that the TA signs is recent?**

3408 It doesn't. EntityA checks however to ensure that the TSN is the same in all TSR. This  
3409 ensures that the values are all related. If TSR1 is an old value then the HashTimeDelta will  
3410 be a large value and the uncertainty of the relation of the signing to the UTC will be large.

#### 3411 **Why does associating a UTC time with a TSV take two steps?**

3412 This is because it takes some time between when a request goes to a time authority and  
3413 when the response comes. The protocol measures this time and uses it to create the time  
3414 deltas. The relationship of TSV to UTC is somewhere between the request and response.

#### 3415 **Affect of power on the tick counter**

3416 As the TPM is not required to maintain an internal clock and battery, how the platform  
3417 provides power to the TPM affects the ability to maintain the tick counter. The original  
3418 mechanism had the TPM maintaining an indication of how the platform provided the power.  
3419 Previous performance does not predict what might occur in the future, as the platform may  
3420 be unable to continue to provide the power (dead battery, pulled plug from wall etc). With  
3421 the knowledge that the TPM cannot accurately report the future, the specification deleted  
3422 tick type from the TPM.

3423 The information relative to what the platform is doing to provide power to the TPM is now a  
3424 responsibility of the TSS. The TSS should first determine how the platform was built, using  
3425 the platform credential. The TSS should also attempt to determine the actual performance  
3426 of the TPM in regards to maintaining the tick count. The TSS can help in this determination  
3427 by keeping track of the tick nonce. The tick nonce changes each time the tick count is lost.  
3428 By comparing the tick nonce across system events the TSS can obtain a heuristic that  
3429 represents how the platform provides power to the TPM.

3430 The TSS must define a standard set of values as to when the tick nonce continues to  
3431 increment across system events.

3432 The following are some PC implementations that give the flavor of what is possible regarding  
3433 the clock on a specific platform.

3434 TICK\_INC - No TPM power battery. Clock comes from PCI clock, may stop from time to time  
3435 due to clock stopping protocols such as CLKRUN.

3436 TICK\_POWER - No TPM power battery. Clock source comes from PCI clock, always runs  
3437 except in S3+.

3438 TICK\_STSTATE - External power (might be battery) consumed by TPM during S3 only. Clock  
3439 source comes either from a system clock that runs during S3 or from crystal/internal TPM  
3440 source.

3441 TICK\_STCLEAR - Standby power used to drive counter. In desktop, may be related to when  
3442 system is plugged into wall. Clock source comes either from a system clock that runs when  
3443 standby power is available or from crystal/internal TPM source.



542

3444TICK\_ALWAYS - TPM power battery. Clock source comes either from a battery powered  
3445system clock that crystal/internal TPM source.

3446**End of informative comment**

## 3447 **21. Context Management**

### 3448 **Start of informative comment**

3449 The TPM is a device that contains limited resources. Caching of the resources may occur  
3450 without knowledge or assistance from the application that loaded the resource. In version  
3451 1.1 there were two types of resources that had need of this support keys and authorization  
3452 sessions. Each type had a separate load and restore operation. In version 1.2 there is the  
3453 addition of transport sessions. To handle these situations generically 1.2 is defining a single  
3454 context manager that all types of resources may use.

3455 The concept is simple, a resource manager requests that wrapping of a resource in a  
3456 manner that securely protects the resource and only allows the restoring of the resource on  
3457 the same TPM and during the same operational cycle.

3458 Consider a key successfully loaded on the TPM. The parent keys that loaded the key may  
3459 have required a different set of PCR registers than are currently set on the TPM. For  
3460 example, the end result is to have key5 loaded. Key3 is protected by key2, which is  
3461 protected by key1, which is protected by the SRK. Key1 requires PCR1 to be in a certain  
3462 state, key2 requires PCR2 to load and key3 requires PCR3. Now at some point in time after  
3463 key1 loaded key2, PCR1 was extended with additional information. If key3 is evicted then  
3464 there is no way to reload key3 until the platform is rebooted. To avoid this type of problem  
3465 the TPM can execute context management routines. The context management routines save  
3466 key3 in its current state and allow the TPM to restore the state without having to use the  
3467 parent keys (key1 and key2).

3468 There are numerous issues with performing context management on sessions. These issues  
3469 revolve around the use of the nonces in the session. If an attacker can successfully store,  
3470 attack, fail and then reload the session the attacker can repeat the attack many times.

3471 The key that the TPM uses to encrypt blobs may be a volatile or non-volatile key. One  
3472 mechanism would be for the TPM to generate a new key on each TPM\_Startup command.  
3473 Another would be for the TPM to generate the key and store it persistently in the  
3474 TPM\_PERMANENT\_DATA area.

3475 The symmetric key should be relatively the same strength as a 2048-bit RSA key. 128-bit  
3476 AES would be appropriate.

### 3477 **End of informative comment**

3478 1. Context management is a required function.

3479 2. Execution of the context commands MUST NOT cause the exposure of any TPM shielded  
3480 location.

3481 3. The TPM MUST NOT allow the context saving of the EK or the SRK.

3482 4. The TPM MAY use either symmetric or asymmetric encryption. For asymmetric  
3483 encryption the TPM MUST use a 2048 RSA key.

3484 5. A wrapped session blob MUST only be loadable once. A wrapped key blob MAY be  
3485 reloadable.

3486 6. The TPM MUST support a minimum of 16 concurrent saved contexts other than keys.  
3487 There is no minimum or maximum number of concurrent saved key contexts.

551

34887. All external session blobs (of type TPM\_RT\_TRANS or TPM\_RT\_AUTH) can be invalidated  
3489 upon specific request (via TPM\_FlushXXX using TPM\_RT\_CONTEXT as resource type).  
3490 This does not include saved context blobs of type TPM\_RT\_KEY.

34918. External session blobs are invalidated on TPM\_Startup(ST\_CLEAR) or on  
3492 TPM\_Startup(any) based on the startup effects settings

3493 a. Saved context blobs of type TPM\_RT\_KEY with the attributes of parentPCRStatus =  
3494 FALSE and isVolatile = FALSE SHOULD not be invalidated on TPM\_Startup(any)

34959. All external sessions invalidate automatically upon installation of a new owner due to the  
3496 setting of a new tpmProof.

349710. If the TPM enters failure mode ALL session blobs (including keys) MUST be invalidated

3498 a. Invalidation includes ensuring that contextNonceKey and contextNonceSession will  
3499 change when the TPM recovers from the failure.

350011. Attempts to restore a wrapped blob after the successful completion of  
3501 TPM\_Startup(ST\_CLEAR) MUST fail. The exception is a wrapped key blob which may be  
3502 long-term and which MAY restore after a TPM\_Startup(ST\_CLEAR).

350312. The save and load context commands are the generic equivalent to the context  
3504 commands in 1.1. Version 1.2 deprecates the following commands:

3505 a. TPM\_AuthSaveContext

3506 b. TPM\_AuthLoadContext

3507 c. TPM\_KeySaveContext

3508 d. TPM\_KeyLoadContext

## 3509 **22. Eviction**

### 3510 **Start of informative comment**

3511 The TPM has numerous resources held inside of the TPM that may need eviction. The need  
3512 for eviction occurs when the number of resources in use by the TPM exceed the available  
3513 space. For resources that are hard to reload (i.e. keys tied to PCR values) the outside entity  
3514 should first perform a context save before evicting items.

3515 In version 1.1 there were separate commands to evict separate resource types. This new  
3516 command set uses the resource types defined for context saving and creates a generic  
3517 command that will evict all resource types.

### 3518 **End of informative comment**

3519 1. The TPM MUST NOT flush the EK or SRK using this command.

3520 2. Version 1.2 deprecates the following commands:

- 3521 a. TPM\_Terminate\_Handle
- 3522 b. TPM\_EvictKey
- 3523 c. TPM\_Reset

## 3524**23. Session pool**

### 3525**Start of informative comment**

3526The TPM supports two types of sessions that use the rolling nonce protocol, authorization  
3527and transport. These sessions require much of the same handling and internal storage by  
3528the TPM. To allow more flexibility the internal storage for these sessions will be defined as  
3529coming from the same pool (or area).

3530The pool requires that three (3) sessions be available. The entities using the TPM can  
3531determine the usage models of what sessions are active. This allows a TPM to have 3  
3532authorization sessions or 3 transport sessions at one time.

3533Using all available pool resources for transport sessions is not a very usable model. If all  
3534resources are in use by transport, there are no resources available for authorization  
3535sessions and hence no ability to execute any commands requiring authorization. A more  
3536realistic model would be to have two transport sessions and one authorization session.  
3537While this is an unrealistic model for actual execution there will be no requirement that the  
3538TPM prevent this from happening. A model of how it could occur would be when there are  
3539two applications running, both using 2 transport sessions and one authorization session.  
3540When switching between the applications, if the requirement was that only 2 transport  
3541sessions could be active the TSS that would provide the context switch would have to  
3542ensure that the transport sessions were context saved first.

3543Sessions can be virtualized, so while the TPM may only have 3 loaded sessions, there may  
3544be an unlimited number of context saved sessions stored outside the TPM.

### 3545**End of informative comment**

35461. The TPM MUST support a minimum of three (3) concurrent sessions. The sessions MAY  
3547 be any mix of authentication and transport sessions.

## 3548 **24. Initialization Operations**

### 3549 **Start of informative comment**

3550 Initialization is the process where the TPM establishes an operating environment from a no  
3551 power state. Initialization occurs in many different flavors with PCR, keys, handles, sessions  
3552 and context blobs all initialized, reloaded or unloaded according to the rules and platform  
3553 environment.

3554 Initialization does not affect the operational characteristics of the TPM (like TPM  
3555 Ownership).

3556 Clear is the process of returning the TPM to factory defaults. The clear commands need  
3557 protection from unauthorized use and must allow for the possibility of changing Owners.  
3558 The clear process requires authorization to execute and locks to prevent unauthorized  
3559 operation.

3560 The clear functionality performs the following tasks:

3561 Invalidate SRK. Invalidating the SRK invalidates all protected storage areas below the SRK  
3562 in the hierarchy. The areas below are not destroyed they just have no mechanism to be  
3563 loaded anymore.

3564 All TPM volatile and non-volatile data is set to default value except the endorsement key  
3565 pair. The clear includes the Owner-AuthData, so after performing the clear, the TPM has no  
3566 Owner. The PCR values are undefined after a clear operation.

3567 The TPM shall return TPM\_NOSRK until an Owner is set. After the execution of the clear  
3568 command, the TPM must go through a power cycle to properly set the PCR values.

3569 The Owner has ultimate control of when a clear occurs.

3570 The Owner can perform the TPM\_OwnerClear command using the TPM Owner  
3571 authorization. If the Owner wishes to disable this clear command and require physical  
3572 access to perform the clear, the Owner can issue the TPM\_DisableOwnerClear command.

3573 During the TPM startup processing anyone with physical access to the machine can issue  
3574 the TPM\_ForceClear command. This command performs the clear. The  
3575 TPM\_DisableForceClear disables the TPM\_ForceClear command for the duration of the  
3576 power cycle. TSS startup code that does not issue the TPM\_DisableForceClear leaves the  
3577 TPM vulnerable to a denial of service attack. The assumption is that the TSS startup code  
3578 will issue the TPM\_DisableForceClear on each power cycle after the TSS determines that it  
3579 will not be necessary to issue the TPM\_ForceClear command. The purpose of the  
3580 TPM\_ForceClear command is to recover from the state where the Owner has lost or  
3581 forgotten the TPM Ownership token.

3582 The TPM\_ForceClear must only be possible when the issuer has physical access to the  
3583 platform. The manufacturer of a platform determines the exact definition of physical access.

### 3584 **End of informative comment**

3585 1. The TPM MUST support proper initialization. Initialization MUST properly configure the  
3586 TPM to execute in the platform environment.

3587 2. Initialization MUST ensure that handles, keys, sessions, context blobs and PCR are  
3588 properly initialized, reloaded or invalidated according to the platform environment.

569

35893. The description of the platform environment arrives at the TPM in a combination of  
3590 TPM\_Init and TPM\_Startup.

## 3591 **25. HMAC digest rules**

### 3592 **Start of informative comment**

3593 The order of calculation of the HMAC is critical to being able to validate the authorization  
3594 and parameters of a command. All commands use the same order and format for the  
3595 calculation.

3596 A more exact representation of a command would be the following

```
3597 *****
3598 * TAG | LEN | ORD | HANDLES | DATA | AUTH1 (o) | AUTH2 (o) *
3599 *****
```

3600 The text area for the HMAC calculation would be the concatenation of the following:

3601 ORD || DATA

### 3602 **End of informative comment**

3603 The HMAC digest of parameters uses the following order

3604 1. Skip tag and length

3605 2. Include ordinal. This is the 1S parameter in the HMAC column for each command

3606 3. Skip handle(s). This includes key and other session handles

3607 4. Include data and other parameters for the command. This starts with the 2S parameter  
3608 in the HMAC column for each command.

3609 5. Skip all AuthData values.



## 3610**26. Generic authorization session termination rules**

### 3611**Start of informative comment**

3612These rules are the generic rules that govern all authorization sessions, a specific session  
3613type may have additional rules or modifications of the generic rules

### 3614**End of informative comment**

36151. A TPM SHALL unilaterally perform the actions of TPM\_FlushSpecific for a session upon  
3616 any of the following events

- 3617 a. “continueUse” flag in the authorization session is FALSE
- 3618 b. Shared secret of the session in use to create the exclusive-or for confidentiality of  
3619 data. Example is TPM\_ChangeAuth terminates the authorization session.  
3620 TPM\_ExecuteTransport does not terminate the session due to protections inherent in  
3621 transport sessions.
- 3622 c. When the associated entity is invalidated
- 3623 d. When the command returns a fatal error. This is due to error returns not setting a  
3624 nonceEven. Without a new nonceEven the rolling nonces sequence is broken hence  
3625 the TPM MUST terminate the session.
- 3626 e. Failure of an authorization check at the start of the command
- 3627 f. Execution of TPM\_Startup(ST\_CLEAR)

36282. The TPM MAY perform the actions of TPM\_FlushSpecific for a session upon the following  
3629 events

- 3630 a. Execution of TPM\_Startup(ST\_STATE)

## 3631 **27. PCR Grand Unification Theory**

### 3632 **Start of informative comment**

3633 This section discusses the unification of PCR definition and use with locality.

3634 The PCR allow the definition of a platform configuration. With the addition of locality, the  
3635 meaning of a configuration is somewhat larger. This section defines how the two combine to  
3636 provide the TPM user information relative to the platform configuration.

3637 These are the issues regarding PCR and locality at this time

### 3638 **Definition of configuration**

3639 A configuration is the combination of PCR, PCR attributes and the locality.

### 3640 **Passing the creators configuration to the user of data**

3641 For many reasons, from the creator's viewpoint and the user's viewpoint, the configuration  
3642 in use by the creator is important information. This information needs transmitting to the  
3643 user with the data and with integrity.

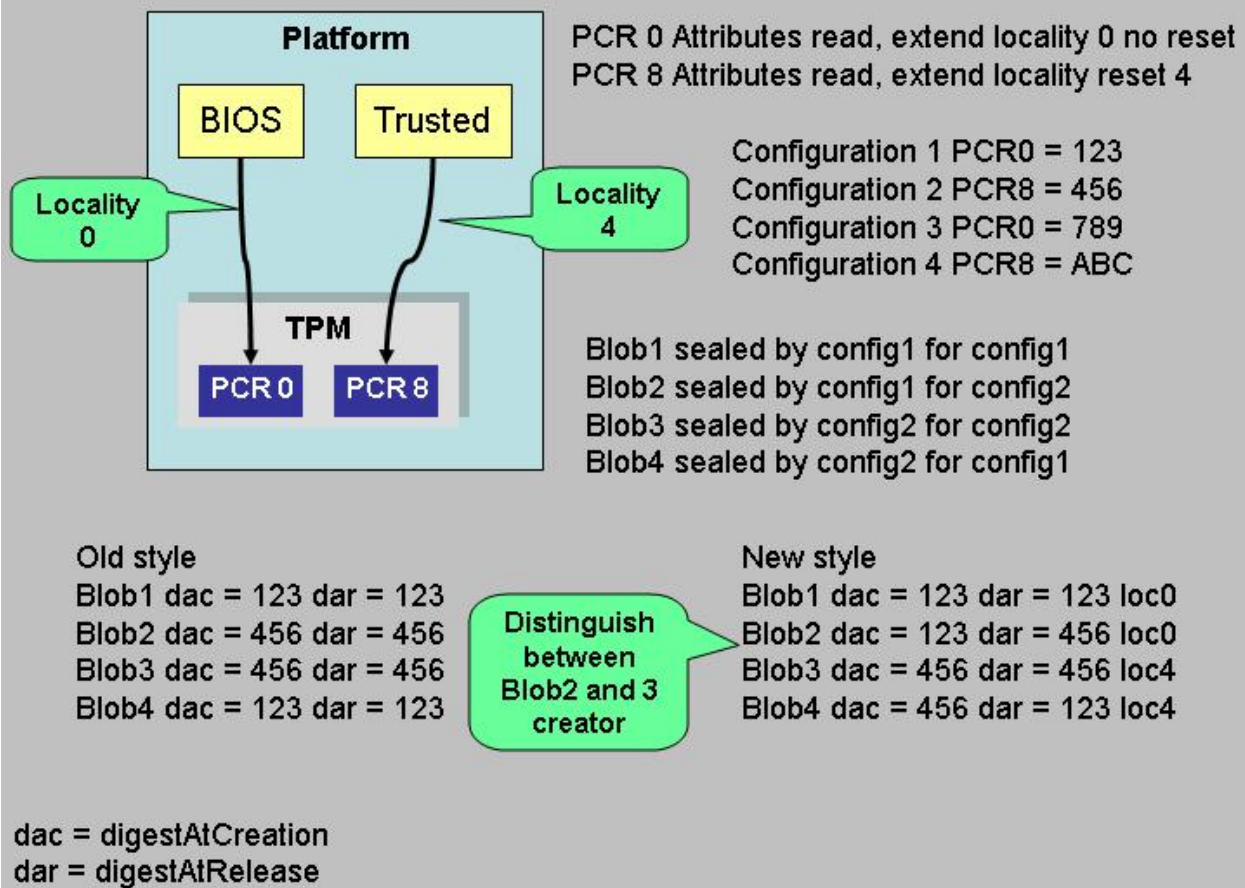
3644 The configuration must include the locality and may not be the same configuration that will  
3645 use the data. This allows one configuration to seal a value for future use and the end user  
3646 to know the genealogy of where the data comes from.

### 3647 **Definition of "Use"**

3648 See the definition of TPM\_PCR\_ATTRIBUTES for the attributes and the normative  
3649 statements regarding the use of the attributes. The use of a configuration is when the TPM  
3650 needs to ensure that the proper platform configuration is present. The first example is for  
3651 Unseal, the TPM must only release the information sealed if the platform configuration  
3652 matches the configuration specified by the seal creator. Here the use of locality is implicit in  
3653 the PCR attributes, if PCR8 requires locality 2 to be present then the seal creator ensures  
3654 that locality 2 is asserted by defining a configuration that uses PCR8.

3655 The creation of a blob that specifies a configuration for use is not a "use" itself. So the SEAL  
3656 command does is not a use for specifying the use of a PCR configuration.

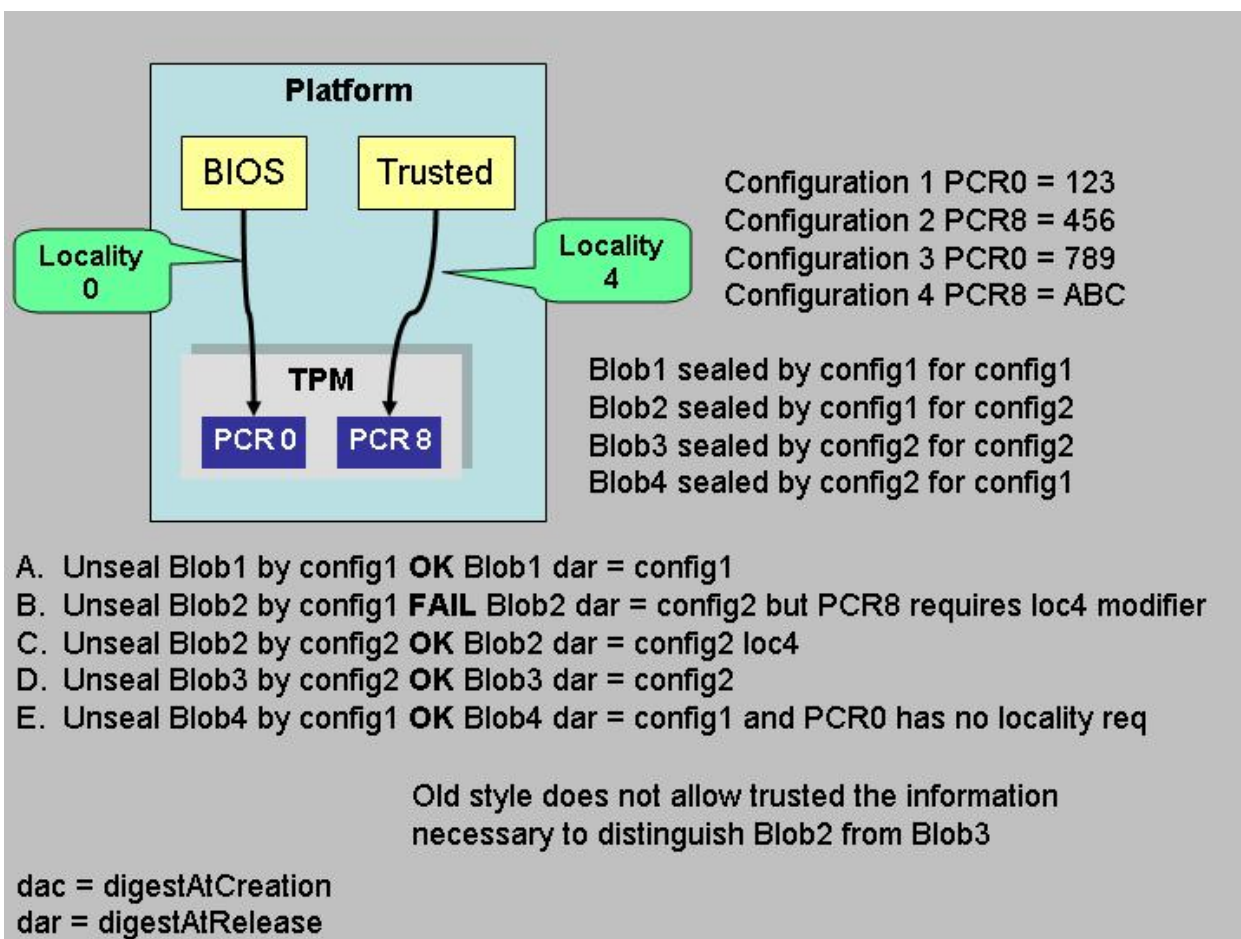
3657



3658

3659 By using the “new style” or TPM\_PCR\_INFO\_LONG structure the user can determine that

3660 Blob2 is different that Blob3.



3661  
3662 Case B is the only failure and this shows the use of the locality modifier and PCR locality  
3663 attribute.

3664 Additional attempts are obvious failures, config3 and config4 are unable to unseal any of  
3665 the 4 blobs.

3666 One example is illustrative of the problems of just specifying locality without an  
3667 accompanying PCR. Assume Blob5 which specifies a dar of config1 and a locality 4 modifier.  
3668 Now either config2 or config4 can unseal Blob5. In fact there is no way to restrict ANY  
3669 process that gains access to locality 4 from performing the unseal. As many platforms will  
3670 have no restrictions as to which process can load in locality 4 there is no additional benefit  
3671 of specifying a locality modifier. If the sealer wants protections, they need to specify a PCR  
3672 that requires a locality modifier.

### 3673 Defining locality modifiers dynamically

3674 This feature would enable the platform to specify how and when a locality modifier applies  
3675 to a PCR. The current definition of PCR attributes has the values set in TPM manufacturing  
3676 and static for all TPM in a specific platform type (like a PC).

3677 Defining dynamic attributes would make the use of a PCR very difficult. The sealer would  
3678 have to have some way of ensuring that their wishes were enforced and challengers would  
3679 have to pay close attention to the current PCR attributes. For these reasons the setting of  
3680 the PCR attributes is defined as a static operation made during the platform specific  
3681 specification.

**3682End of informative comment****368327.1 Validate Key for use****3684Start of informative comment**

3685The following shows the order and checks done before the use of a key that has PCR or  
3686locality restrictions.

3687Note that there is no check for the PCR registers on the DSAP session. This is due to the  
3688fact that DSAP checks for the continued validity of the PCR that are attached to the DSAP  
3689and any change causes the invalidation of the DSAP session.

3690The checks must validate the locality of the DSAP session as the PCR registers in use could  
3691have locality restrictions.

**3692End of informative comment**

36931. If the authorization session is DSAP

3694 a. If the DSAP -> localityAtRelease is not 0x1F (or in other words some localities are not  
3695 allowed)

3696 i. Validate that TPM\_STANY\_FLAGS -> localityModifier is matched by DSAP ->  
3697 pcrInfo -> localityAtRelease, on mismatch return TPM\_BAD\_LOCALITY

3698 b. If DSAP -> digestAtRelease is not 0

3699 i. Calculate the current digest and compare to digestAtRelease, return  
3700 TPM\_BAD\_PCR on mismatch

3701 c. If the DSAP points to an ordinal delegation

3702 i. Check that the DSAP authorizes the use of the intended ordinal

3703 d. If the DSAP points to a key delegation

3704 i. Check that the DSAP authorizes the use of the key

3705 e. If the key delegated is a CMK key

3706 i. The TPM MUST check the CMK\_DELEGATE restrictions

37072. Set LK to the loaded key that is being used

37083. If LK -> pcrInfoSize is not 0

3709 a. If LK -> pcrInfo -> releasePCRSelection identifies the use of one or more PCR

3710 i. Calculate H1 a TPM\_COMPOSITE\_HASH of the PCR selected by LK -> pcrInfo ->  
3711 releasePCRSelection

3712 ii. Compare H1 to LK -> pcrInfo -> digestAtRelease on mismatch return  
3713 TPM\_WRONGPCRVAL

3714 b. If localityAtRelease is NOT 0x1F

3715 i. Validate that TPM\_STANY\_FLAGS -> localityModifier is matched by LK -> pcrInfo  
3716 -> localityAtRelease on mismatch return TPM\_BAD\_LOCALITY

37174. Allow use of the key

## 3718**28. Non Volatile Storage**

### 3719**Start of informative comment**

3720The TPM contains protected non-volatile storage. There are many uses of this type of area;  
3721however, a TPM needs to have a defined set of operations that touch any protected area.  
3722The idea behind these instructions is to provide an area that the manufacturers and owner  
3723can use for storing information in the TPM.

3724The TCG will define a limited set of information that it sees a need of storing in the TPM.  
3725The TPM and platform manufacturer may add additional areas.

3726The NV storage area has a limited use before it will no longer operate Hence the NV  
3727commands are under TPM Owner control.

3728Controls exist to allow a manufacturer to define and write NV indexes during  
3729manufacturing before an owner exists. This is strictly a manufacturing mode, as it allows a  
3730manufacturer to bypass security.

3731To locate if an index is available, use TPM\_GetCapability to return the index and the size of  
3732the area in use by the index.

3733The area may not be larger than the TPM input buffer. The TPM will report the maximum  
3734size available to allocate.

3735The storage area is an opaque area to the TPM. The TPM, other than providing the storage,  
3736does not review the internals of the area.

3737To SEAL a blob, the creator of the area specifies the use of PCR registers to read the value.  
3738This is the exact property of SEAL.

3739To obtain a signed indication of what is in a NV store area the caller would setup a  
3740transport session with logging on and then get the signed log. The log shows the parameters  
3741so the caller can validate that the TPM holds the value.

3742There is an attribute, for each index, that defines the expected write scheme for the index.  
3743The TPM may handle data storage differently based on the write scheme attribute that  
3744defines the expected for the index. Whenever possible the NV memory should be allocated  
3745with the write scheme attribute set to update as one block and not as individual bytes.

3746The non-volatile storage described here is defined by TPM\_NV\_DefineSpace. Other  
3747structures that a manufacturer might decide to store in non-volatile memory (e.g., PCRs,  
3748keys, the audit digest) are logically separate and do not affect the space available for the NV  
3749indexed storage described here. An exception is a key that is moved from volatile to NV  
3750memory when set as "owner evict". This NV memory may come from a pool shared with NV  
3751define space.

### 3752**End of informative comment**

37531. The TPM MUST support the NV commands. The TPM MUST support the NV area as  
3754 defined by the TPM\_NV\_INDEX values.

37552. The TPM MAY manage the storage area using any allocation and garbage collection  
3756 scheme.

605

37573. To remove an area from the NV store the TPM owner would use the  
3758 TPM\_NV\_DefineSpace command with a size of 0. Any authorized user can change the  
3759 value written in the NV store.

37604. The TPM MUST treat the NV area as a shielded location.

3761 a. The TPM does not provide any additional protections (like additional encryption) to  
3762 the NV area.

37635. If a write operation is interrupted, then the TPM makes no guarantees about the data  
3764 stored at the specified index. It MAY be the previous value, MAY be the new value or  
3765 MAY be undefined or unpredictable. After the interruption the TPM MAY indicate that  
3766 the index contains unpredictable information.

3767 a. The TPM MUST ensure that in case of interruption of a write to an index that all  
3768 other indexes are not affected

37696. Minimum size of NV area is platform specific. The maximum area is TPM vendor specific.

37707. A TPM MUST NOT use the NV area to store any data dependent on data structures  
3771 defined in Part II of the TPM specifications, except for the NV Storage structures implied  
3772 by required index values or reserved index values.

## 377328.1 NV storage design principles

### 3774Start of informative comment

3775This section lists the design principles that motivate the NV area in the TPM. There was the  
3776realization that the current design made use of NV storage but not necessarily efficiently.  
3777The DIR, BIT and other commands placed demands on the TPM designer and required  
3778areas that while allowing for flexible use reserved space most likely never used (like DIR for  
3779locality 1).

3780The following are the design principles that drive the function definitions.

37811. Provide efficient use of NV area on the TPM. NV storage is a very limited resource and  
3782data stored in the NV area should be as small as possible.

37832. The TPM does not control, edit, validate or manipulate in any manner the information in  
3784the NV store. The TPM is merely a storage device. The TPM does enforce the access rules as  
3785set by the TPM Owner.

37863. Allocation of the NV area for a specific use must be under control of the TPM Owner.

37874. The TPM Owner, when defining the area to use, will set the access and use policy for the  
3788area. The TPM Owner can set AuthData values, delegations, PCR values and other controls  
3789on the access allowed to the area.

37905. There must be a capability to allow TPM and platform manufacturers to use this area  
3791without a TPM Owner being present. This allows the manufacturer to place information into  
3792the TPM without an onerous manufacturing flow. Information in this category would  
3793include EK credential and platform credential.

37946. The management and use of the NV area should not require a large number of ordinals.

37957. The management and use of the NV area should not introduce new operating strategies  
3796into the TPM and should be easy to implement.

3797 **End of informative comment**

## 3798 **28.1.1 NV Storage use models**

3799 **Start of informative comment**

3800 This informative section describes some of the anticipated use models and the attributes a  
3801 user of the storage area would need to set.

3802

3803 **Owner authorized for all access**

3804 TPM\_NV\_DefineSpace: attributes = PER\_OWERREAD || PER\_OWNERWRITE

3805 WriteValue(TPM Owner Auth, data)

3806 ReadValue(TPM Owner Auth, data)

3807

3808 **Set AuthData value**

3809 TPM\_NV\_DefineSpace: attributes = PER\_AUTHREAD || PER\_AUTHWRITE, auth =  
3810 authValue

3811 WriteValue( authValue, data)

3812 ReadValue( authValue, data)

3813

3814 **Write once, only way to change is to delete and redefine**

3815 TPM\_NV\_DefineSpace: attributes = PER\_WRITEDEFINE

3816 WriteValue( size = x, data) // successful

3817 WriteValue(size = 0) // locks

3818 WriteValue(size = x) // fails

3819...

3820 TPM\_Startup(ST\_Clear) // Does not affect lock

3821 WriteValue(size = x, data) // fails

3822

3823 **Write until specific index is locked, lock reset on Startup(ST\_Clear)**

3824 TPM\_NV\_DefineSpace: index = 3, attributes = PER\_WRITE\_STCLEAR

3825 TPM\_NV\_DefineSpace: index = 5, attributes = PER\_WRITE\_STCLEAR

3826 WriteValue( index = 3, size = x, data) // successful

3827 WriteValue( index = 5, size = x, data) // successful

3828 WriteValue(index = 3, size = 0) // locks

3829 WriteValue( index = 3, size = x, data) // fails

3830 WriteValue( index = 5, size = x, data) // successful



614

3831...

3832TPM\_Startup(ST\_Clear) // clears lock

3833WriteValue( index = 3, size = x, data) // successful

3834WriteValue( index = 5, size = x, data) // successful

3835

3836**Write until index 0 is locked, lock reset by Startup(ST\_Clear)**

3837TPM\_NV\_DefineSpace: attributes = PER\_GLOBALLOCK, index = 5

3838TPM\_NV\_DefineSpace: attributes = PER\_GLOBALLOCK, index = 3

3839WriteValue( index = 3, size = x, data) // successful

3840WriteValue( index = 5, size = x, data) // successful

3841

3842WriteValue(index = 0) // sets SV -&gt; bGlobalLock to TRUE

3843WriteValue( index = 3, size = x, data) // fails

3844WriteValue( index = 5, size = x, data) // fails

3845...

3846TPM\_Startup(ST\_Clear) // clears lock

3847WriteValue( index = 3, size = x, data) // successful

3848WriteValue( index = 5, size = x, data) // successful

3849**End of informative comment**

## 3850**28.2 Use of NV storage during manufacturing**

3851**Start of informative comment**

3852The TPM needs the ability to write values to the NV store during manufacturing. It is  
 3853possible that the values written at this time would require authorization during normal TPM  
 3854use. The actual enforcement of these authorizations during manufacturing would cause  
 3855numerous problems for the manufacturer.

3856The TPM will not enforce the NV authorization restrictions until the execution of a  
 3857TPM\_NV\_DefineSpace with the handle of TPM\_NV\_INDEX\_LOCK.

3858The 'D' bit indicates an NV index defined (typically) during manufacturing and then locked.  
 3859While nvLocked is FALSE, indices with the 'D' set can be defined, deleted, or redefined as  
 3860desired. Once nvLocked is set TRUE, the 'D' bit indices are locked. They cannot be defined,  
 3861deleted or redefined.

3862nvLocked has the lifetime of the endorsement key.

3863**End of informative comment**

38641. The TPM MUST NOT enforce the NV authorizations (auth values, PCR etc.) prior to the  
 3865 execution of TPM\_NV\_DefineSpace with an index of TPM\_NV\_INDEX\_LOCK

- 3866 a. While the TPM is not enforcing NV authorizations, the TPM SHALL allow the use of  
3867 TPM\_NV\_DefineSpace in any operational state (disabled, deactivated)

## 3868**29. Delegation Model**

### 3869**Start of informative comment**

3870The TPM Owner is an entity with a single “super user” privilege to control TPM operation.  
3871Thus if any aspect of a TPM requires management, the TPM Owner must perform that task  
3872himself or reveal his privilege information to another entity. This other entity thereby  
3873obtains the privilege to operate all TPM controls, not just those intended by the Owner.  
3874Therefore the Owner often must have greater trust in the other entity than is strictly  
3875necessary to perform an arbitrary task.

3876This delegation model addresses this issue by allowing delegation of individual TPM Owner  
3877privileges (the right to use individual Owner authorized TPM commands) to individual  
3878entities, which may be trusted processes.

3879Basic requirements:

3880**Consumer user does not need to enter or remember a TPM Owner password.** This is an  
3881ease of use and security issue. Not remembering the password may lead to bad security  
3882practices, increased tech support calls and lost data.

3883**Role based administration and separation of duty.** It should be possible to delegate just  
3884enough Owner privileges to perform some administration task or carry out some duty,  
3885without delegating all Owner privileges.

3886**TPM should support multiple trusted processes.** When a platform has the ability to load  
3887and execute multiple trusted processes then the TPM should be able to participate in the  
3888protection of secrets and proper management of the processes and their secrets. In fact, the  
3889TPM most likely is the root of storage for these values. The TPM should enable the proper  
3890management, protection and distribution of values held for the various trusted processes  
3891that reside on the same platform.

3892**Trusted processes may require restrictions.** A fundamental security tenet is the principle  
3893of least privilege, that is, to limit process functionality to only the functions necessary to  
3894accomplish the task. This delegation model provides a building block that allows a system  
3895designer to create single purpose processes and then ensure that the process only has  
3896access to the functions that it requires to complete the task.

3897**Maintain the current authorization structure and protocols.** There is no desire to  
3898remove the current TPM Owner and the protocols that authorize and manage the TPM  
3899Owner. The capabilities are a delegation of TPM Owner responsibilities. The delegation  
3900allows the TPM Owner to delegate some or all of the actions that a TPM Owner can perform.  
3901The TPM Owner has complete control as to when and if the capability delegation is in use.

### 3902**End of informative comment**

## 3903**29.1 Table Requirements**

### 3904**Start of informative comment**

3905**No ocean front property in table** – We want the table to be virtually unlimited in size.  
3906While we need some storage, we do not want to pick just one number and have that be the  
3907min and max. This drives the need for the ability to save, off the TPM, delegation elements.

3908 **Revoking a delegation, does not affect other** delegations – The TPM Owner may, at any  
3909 time, determine that a delegation is no longer appropriate. The TPM Owner needs to be able  
3910 to ensure the revocation of all delegations in the same family. The TPM Owner also wants to  
3911 ensure that revocation done in one family does not affect any other family of delegations.

3912 **Table seeded by OEM** – The OEM should do the seeding of the table during manufacturing.  
3913 This allows the OEM to ship the platform and make it easy for the platform owner to  
3914 startup the first time. The definition of manufacturing in this context includes any time  
3915 prior to or including the time the user first turns on the platform.

3916 **Table not tied to a TPM owner** – The table is not tied to the existence of a TPM owner. This  
3917 facilitates the seeding of the table by the OEM.

3918 **External delegations need authorization and assurance of** revocation – When a  
3919 delegation is held external to the TPM, the TPM must ensure authorization of the delegation  
3920 when loading the delegation. Upon revocation of a family or other family changes the TPM  
3921 must ensure that prior valid delegations are not successfully loaded.

3922 **90% case, no need for external store** – The normal case should be that the platform does  
3923 not need to worry about having external delegations. This drives the need for some NV  
3924 storage to hold a minimum number of table rows.

3925 **End of informative comment**

## 3926 **29.2 How this works**

3927 **Start of informative comment**

3928 The existing TPM owner authorization model is that certain TPM commands require the  
3929 authorization of the TPM Owner to operate. The authorization value is the TPM Owners  
3930 token. Using the token to authorize the command is proof of TPM Ownership. There is only  
3931 one token and knowledge of this token allows all operations that require proof of TPM  
3932 Ownership.

3933 This extension allows the TPM Owner to create a new AuthData value and to delegate some  
3934 of the TPM Ownership rights to the new AuthData value.

3935 The use model of the delegation is to create an authorization session (DSAP) using the  
3936 delegated AuthData value instead of the TPM Owner token. This allows delegation to work  
3937 without change to any current command.

3938 The intent is to permit delegation of selected Owner privileges to selected entities, be they  
3939 local or remote, separate from the current software environment or integrated into the  
3940 current software environment. Thus Owner privileges may be delegated to entities on other  
3941 platforms, to entities (trusted processes) that are part of the normal software environment  
3942 on the Owner's platform, or to a minimalist software environment on the Owner's platform  
3943 (created by booting from a CDROM, or special disk partition), for example.

3944 Privileges may be delegated to a particular entity via definition of a particular process on the  
3945 Owner's platform (by dictating PCR values), and/or by stipulating a particular AuthData  
3946 value. The resultant TPM\_DELEGATE\_OWNER\_BLOB and any AuthData value must be  
3947 passed by the Owner to the chosen entity.

3948 Delegation to an external entity (not on the Owner's platform) probably requires an  
3949 AuthData value and a NULL PCR selection. (But the AuthData value might be sealed to a  
3950 desired set of PCRs in that remote platform.)

3951Delegation to a trusted process provided by the local OS requires a PCR that indicates the  
3952trusted process. The authorization token should be a fixed value (any well known value),  
3953since the OS has no means to safely store the authorization token without sealing that  
3954token to the PCR that indicates the trusted process. It is suggested that the value 0x111...  
3955111 be used.

3956Delegation to a specially booted entity requires either a PCR or an authorization token, and  
3957preferably both, to recognize both the process and the fact that the Owner wishes that  
3958process to execute.

3959The central delegation data structure is a set of tables. These tables indicate the command  
3960ordinals delegated by the TPM Owner to a particular defined environment. The tables allow  
3961the distinction of delegations belonging to different environments.

3962The TPM is capable of storing internally a few table elements to enable the passing of the  
3963delegation information from an entity that has no access to memory or storage of the  
3964defined environment.

3965The number of delegations that the tables can hold is a dynamic number with the  
3966possibility of adding or deleting entries at any time. As the total number is dynamic, and  
3967possibly large, the TPM provides a mechanism to cache the delegations. The cache of a  
3968delegation must include integrity and confidentiality. The term for the encrypted cached  
3969entity is blob. The blob contains a counter (verificationCount) validated when the TPM loads  
3970the blob.

3971An Owner uses the counter mechanism to prevent the use of undesirable blobs; they  
3972increment verificationCount inside the TPM and insert the current value of  
3973verificationCount into selected table elements, including temporarily loaded blobs. (This is  
3974the reason why a TPM must still load a blob that has an incorrect verificationCount.) An  
3975Owner can verify the delegation state of his platform (immediately after updating  
3976verificationCount) by keeping copies of the elements that have just been given the current  
3977value of verificationCount, signing those copies, and sending them to a third party.

3978Verification probably requires interaction with a third party because acceptable table  
3979profiles will change with time and the most important reason for verification is suspicion of  
3980the state of a TOS in a platform. Such suspicion implies that the verification check must be  
3981done by a trusted security monitor (perhaps separate trusted software on another platform  
3982or separate trusted software on CDROM, for example). The signature sent to the third party  
3983must include a freshness value, to prevent replay attacks, and the security monitor must  
3984verify that a response from the third party includes that freshness value. In situations  
3985where the highest confidence is required, the third party could provide the response by an  
3986out-of-band mechanism, such as an automated telephone service with spoken confirmation  
3987of acceptability of platform state and freshness value.

3988A challenger can verify an entire family using a single transport session with logging, that  
3989increments the verification count, updates the verification count in selected blobs, reads the  
3990tables and obtains a single transport session signature over all of the blobs in a family.

3991If no Owner is installed, the delegation mechanisms are inoperative and third party  
3992verification of the tables is impossible, but tables can still be administered and corrected.  
3993(See later for more details.)

3994To perform an operation using the delegation the entity establishes an authorization session  
3995and uses the delegated AuthData value for all HMAC calculations. The TPM validates the

3996AuthData value, and in the case of defined environments checks the PCR values. If the  
3997validation is successful, the TPM then validates that the delegation allows the intended  
3998operation.

3999There can be at least two delegation rows stored in non-volatile storage inside a TPM, and  
4000these may be changed using Owner privilege or delegated Owner privilege. Each delegation  
4001table row is a member of a family, and there can be at least eight family rows stored in non-  
4002volatile storage inside a TPM. An entity belonging to one family can be delegated the  
4003privilege to create a new family and edit the rows in its own family, but no other family.

4004In addition to tying together delegations, the family concept and the family table also  
4005provides the mechanism for validation and revocation of exported delegate table rows, as  
4006well as the mechanism for the platform user to perform validation of all delegations in a  
4007family.

4008**End of informative comment**

## 4009**29.3 Family Table**

4010**Start of informative comment**

4011The family table has three main purposes.

40121 - To provide for the grouping of rows in the TPM\_DELEGATE\_TABLE; entities identified in  
4013delegate table rows as belonging to the same family can edit information in the other  
4014delegate table rows with the same family ID. This allows a family to manage itself and  
4015provides an easier mechanism during upgrades.

40162 - To provide the validation and revocation mechanism for exported  
4017TPM\_DELEGATE\_ROWS and those stored on the TPM in the delegation table

40183 - To provide the ability to perform validation of all delegations in a family

4019The family table must have eight rows, and may have more. The maximum number of rows  
4020is TPM vendor-defined and is available using the TPM\_GetCapability command.

4021As the family table has a limited number of rows, there is the possibility that this number  
4022could be insufficient. However, the ability to create a virtual amount of rows, like done for  
4023the TPM\_DELEGATE\_TABLE would create the need to have all of the validation and  
4024revocation mechanisms that the family table provides for the delegate table. This could  
4025become a recursive process, so for this version of the specification, the recursion stops at  
4026the family table.

4027The family table contains four pieces of information: the family ID, the family label, the  
4028family verification count, and the family flags.

4029The family ID is a 32-bit value that provides a sequence number of the families in use.

4030The family label is a one-byte field that family table manager software would use to help  
4031identify the information associated with the family. Software must be able to map the  
4032numeric value associated with each family to the ASCII-string family name displayable in  
4033the user interface.

4034The family verification count is a 32-bit sequence number that identifies the last outside  
4035verification and attestation of the family information.

4036Initialization of the family table occurs by using the TPM\_Delegate\_Manage command with  
4037the TPM\_FAMILY\_CREATE option.

4038The verificationCount parameter enables a TPM to check that all rows of a family in the  
4039delegate table are approved (by an external verification process), even if rows have been  
4040stored off-TPM.

4041The family flags allow the use and administration of the family table row, and its associated  
4042delegate table rows.

#### 4043**Row contents**

4044Family ID – 32-bits

4045Row label – One byte

4046Family verification count – 32-bits

4047Family enable/disable use/admin flags – 32-bits

#### 4048**End of informative comment**

## 4049**29.4 Delegate Table**

#### 4050**Start of informative comment**

4051The delegate table has three main purposes, from the point of view of the TPM. This table  
4052holds:

4053The list of ordinals allowable for use by the delegate

4054The identity of a process that can use the ordinal list

4055The AuthData value to use the ordinal list

4056The delegate table has a minimum of two (2) rows; the maximum number of rows is TPM  
4057vendor-defined and is available using the TPM\_GetCapability command. Each row  
4058represents a delegation and, optionally, an assignment of that delegation to an identified  
4059trusted process.

4060The non-volatile delegate rows permit an entity to pass delegation rows to a software  
4061environment without regard to shared memory between the entity and the software  
4062environment. The size of the delegate table does not restrict the number of delegations  
4063because TPM\_Delegate\_CreateOwnerDelegation can create blobs for use in a DSAP session,  
4064bypassing the delegate table.

4065The TPM Owner controls the tables that control the delegations, but (recursively) the TPM  
4066Owner can delegate the management of the tables to delegated entities. Entities belonging  
4067to a particular group (family) of delegation processes may edit delegate table entries that  
4068belong to that family.

4069After creation of a delegation entry there is no restriction on the use of the delegation in a  
4070properly authorized session. The TPM Owner has properly authorized the creation of the  
4071delegation so the use of the delegation occurs whenever the delegate wishes to use it.

4072The rows of the delegate table held in non-volatile storage are only changeable under TPM  
4073Owner authorization.

4074The delegate table contains six pieces of information: PCR information, the AuthData value  
4075for the delegated capabilities, the delegation label, the family ID, the verification count, and

4076 a profile of the capabilities that are delegated to the trusted process identified by the PCR  
4077 information.

#### 4078 **Row Elements**

4079 ASCII label – Label that provides information regarding the row. This is not a sensitive item.

4080 Family ID – The family that the delegation belongs to; this is not a sensitive item.

4081 Verification count – Specifies the version, or generation, of this row; version validity  
4082 information is in the family table. This is not a sensitive value.

4083 Delegated capabilities – The capabilities granted, by the TPM Owner, to the identified  
4084 process. This is not a sensitive item.

#### 4085 **Authorization and Identity**

4086 The creator of the delegation sets the AuthData value and the PCR selection. The creator is  
4087 responsible for the protection and dissemination of the AuthData value. This is a sensitive  
4088 value.

#### 4089 **End of informative comment**

4090 1. The TPM\_DELEGATE\_TABLE MUST have at least two (2) rows; the maximum number of  
4091 table rows is TPM-vendor defined and MUST be reported in response to a  
4092 TPM\_GetCapability command

4093 2. The AuthData value and the PCR selection must be set by the creator of the delegation

## 4094 **29.5 Delegation Administration Control**

#### 4095 **Start of informative comment**

4096 The delegate tables (both family and delegation) present some control problems. The tables  
4097 must be initialized by the platform OEM, administered and controlled by the TPM Owner,  
4098 and reset on changes of TPM Ownership. To provide this level of control there are three  
4099 phases of administration with different functions available in the phases.

4100 The three phases of table administration are; manufacturing (P1), no-owner (P2) and owner  
4101 present (P3). These three phases allow different types of administration of the delegation  
4102 tables.

#### 4103 **Manufacturing (P1)**

4104 A more accurate definition of this phase is open, un-initialized and un-owned. It occurs  
4105 after TPM manufacturing and as a result of TPM\_OwnerClear or TPM\_ForceClear.

4106 In P1 TPM\_Delegate\_Manage can initialize and manage non-volatile family rows in the TPM.  
4107 TPM\_Delegate\_LoadOwnerDelegation can load non-volatile delegation rows in the TPM.

4108 Attacks that attempt to burnout the TPM's NV storage are frustrated by the NV store's own  
4109 limits on the number of writes when no Owner is installed.

#### 4110 **No-Owner (P2)**

4111 This phase occurs after the platform has been properly setup. The setup can occur in the  
4112 platform manufacturing flow, during the first boot of the platform or at any time when the  
4113 platform owner wants to lock the table settings down. There is no TPM Owner at this time.



4114TPM\_Delegate\_Manage locks both the family and delegation rows. This lock can be opened  
4115only by the Owner (after the Owner has been installed, obviously) or by the act of removing  
4116the Owner (even if no Owner is installed). Thus locked tables can be unlocked by asserting  
4117Physical Presence and executing TPM\_ForceClear, without having to install an Owner.

4118In P2, the relevant TPM\_Delegate\_xxx commands all return the error  
4119TPM\_DELEGATE\_LOCKED. This is not an issue as there is no TPM Owner to delegate  
4120commands, so the inability to change the tables or create delegations does not affect the  
4121use of the TPM.

#### 4122**Owned (P3)**

4123In this phase, the TPM has a TPM Owner and the TPM Owner manages the table as the  
4124Owner sees fit. This phase continues until the removal of the TPM Owner.

4125Moving from P2 to P3 is automatic upon establishment of a TPM Owner. Removal of the  
4126TPM Owner automatically moves back to P1.

4127The TPM Owner always has the ability to administer any table. The TPM Owner may  
4128delegate the ability to manipulate a single family or all families. Such delegations are  
4129operative only if delegations are enabled.

#### 4130**End of informative comment**

41311. When DelegateAdminLock is TRUE the TPM MUST disallow any changes to the delegate  
4132 tables

41332. With a TPM Owner installed, the TPM Owner MUST authorize all delegate table changes

### 4134**29.5.1 Control in Phase 1**

#### 4135**Start of informative comment**

4136The TPM starts life in P1. The TPM has no owner and the tables are empty. It is desirable  
4137for the OEM to initialize the tables to allow delegation to start immediately after the Owner  
4138decides to enable delegation. As the setup may require changes and validation, a simple  
4139mechanism of writing to the area once is not a valid option.

4140TPM\_Delegate\_Manage and TPM\_Delegate\_LoadOwnerDelegation allow the OEM to fill the  
4141table, read the public parts of the table, perform reboots, reset the table and when finally  
4142satisfied as to the state of the platform, lock the table.

4143Alternatively, the OEM can leave the tables NULL and turn off table administration leaving  
4144the TPM in an unloaded state waiting for the eventual TPM Owner to fill the tables, as they  
4145need.

4146Flow to load tables

4147Default values of DelegateAdminLock are set either during manufacturing or are the result  
4148of TPM\_OwnerClear or TPM\_ForceClear.

4149TPM\_Delegate\_Manage verifies that DelegateAdminLock is FALSE and that there is no TPM  
4150Owner. The command will therefore load or manipulate the family tables as specified in the  
4151command.

4152TPM\_Delegate\_LoadOwnerDelegation verifies that DelegateAdminLock is FALSE and no TPM  
4153owner is present. The command loads the delegate information specified in the command.

4154 **End of informative comment**

## 4155 **29.5.2 Control in Phase 2**

4156 **Start of informative comment**

4157 In phase 2, no changes are possible to the delegate tables. The platform owner must install  
4158 a TPM Owner and then manage the tables, or use TPM\_ForceClear to revert to phase 1.

4159 **End of informative comment**

## 4160 **29.5.3 Control in Phase 3**

4161 **Start of informative comment**

4162 The TPM\_DELEGATE\_TABLE requires commands that manage the table. These commands  
4163 include filling the table, turning use of the table on or off, turning administration of the  
4164 table on or off, and using the table.

4165 The commands are:

4166 **TPM\_Delegate\_Manage** – Manages the family table on a row-by-row basis: creates a new  
4167 family, enables/disables use of a family table row and delegate table rows that share the  
4168 same family ID, enables/disables administration of a family's rows in both the family table  
4169 and the delegate table, and invalidates an existing family.

4170 **TPM\_Delegate\_CreateOwnerDelegation** increments the family verification count (if  
4171 desired) and delegates the Owner's privilege to use a set of command ordinals, by creating a  
4172 blob. Such blobs can be used as input data for TPM\_DSAP or  
4173 TPM\_Delegate\_LoadOwnerDelegation. Incrementing the verification count and creating a  
4174 delegation must be an atomic operation. Otherwise no delegations are operative after  
4175 incrementing the verification count.

4176 **TPM\_Delegate\_LoadOwnerDelegation** loads a delegate blob into a non-volatile delegate  
4177 table row, inside the TPM.

4178 **TPM\_Delegate\_ReadTable** is used to read from the TPM the public contents of the family  
4179 and delegate tables that are stored on the TPM.

4180 **TPM\_Delegate\_UpdateVerification** sets the verificationCount in an entity (a blob or a  
4181 delegation row) to the current family value, in order that the delegations represented by that  
4182 entity will continue to be accepted by the TPM.

4183 **TPM\_Delegate\_VerifyDelegation** loads a delegate blob into the TPM, and returns success  
4184 or failure, depending on whether the blob is currently valid.

4185 **TPM\_DSAP** – opens a deferred authorization session, using either an input blob (created by  
4186 TPM\_Delegate\_CreateOwnerDelegation) or a cached blob (loaded by  
4187 TPM\_Delegate\_LoadOwnerDelegation into one of the TPM's non-volatile delegation rows).

4188 **End of informative comment**

## 4189 **29.6 Family Verification**

4190 **Start of informative comment**

4191 The platform user may wish to have confirmation that the delegations in use provide a  
4192 coherent set of delegations. This process would require some evaluation of the processes

4193granted delegations. To assist in this confirmation the TPM provides a mechanism to group  
4194all delegations of a family into a signed blob. The signed blob allows the verification agent to  
4195look at the delegations, the processes involved and make an assessment as the validity of  
4196the delegations. The third party then sends back to the platform owner the results of the  
4197assessment.

4198To perform the creation of the signed blob the platform owner needs the ability to group all  
4199of the delegations of a single family into a transport session. The platform owner also wants  
4200an assurance that no management of the table is possible during the verification.

4201This verification does not prove to a third party that the platform owner is not cheating.  
4202There is nothing to prevent the platform owner from performing the validation and then  
4203adding an additional delegation to the family.

4204Here is one example protocol that retrieves the information necessary to validate the rows  
4205belonging to a particular family. Note that the local method of executing the protocol must  
4206prevent a man-in-the-middle attack using the nonce supplied by the user.

4207The TPM Owner can increment the family verification count or use the current family  
4208verification count. Using the current family verification count carries the risk that  
4209unexamined delegation blobs permit undesirable delegations. Using an incremented  
4210verification count eliminates that risk. The entity gathering the verification data requires  
4211Owner authorization or access to a delegation that grants access to transport session  
4212commands, plus other commands depending on whether verificationCount is to be  
4213incremented. This delegation could be a trusted process that can use the delegations  
4214because of its PCR measurements, a remote entity that can use the delegations because the  
4215Owner has sent it a TPM\_DELEGATE\_OWNER\_BLOB and AuthData value, or the host  
4216platform booted from a CDROM that can use the delegations because of its PCR  
4217measurements, and TPM\_DELEGATE\_OWNER\_BLOB and AuthData value submitted by the  
4218Owner, for example.

4219Verification using the current verificationCount

4220The gathering entity requires access to a delegation that grants access to at least the  
4221ordinals to perform a transport session, plus TPM\_Delegate\_ReadTable and  
4222TPM\_Delegate\_VerifyDelegation.

4223The TPM Owner creates a transport session with the “no other activity” attribute set. This  
4224ensures notification if other operations occur on the TPM during the validation process. (If  
4225other operations do occur, the validation processes may have been subverted.) All  
4226subsequent commands listed are performed using the transport session.

4227TPM\_Delegate\_ReadTable displays all public values (including the permissions and PCR  
4228values) in the TPM.

4229TPM\_Delegate\_VerifyDelegation loads each cached blob, with all public values (including the  
4230permissions and PCR values) in plain text.

4231After verifying all blobs, TPM\_ReleaseTransportSigned signs the list of transactions.

4232The gathering entity sends the log of the transport session plus any supporting information  
4233to the validation entity, which evaluates the signed transport session log and informs the  
4234platform owner of the result of the evaluation. This could be an out-of-band process.

4235Verification using an incremented verificationCount

4236 The gathering entity requires Owner authorization or access to a delegation that grants  
4237 access to at least the ordinals to perform a transport session, plus  
4238 TPM\_Delegate\_CreateOwnerDelegation, TPM\_Delegate\_ReadTable, and  
4239 TPM\_Delegate\_UpdateVerification.

4240 The TPM Owner creates a transport session with the “no other activity” attribute set.

4241 To increment the count the TPM Owner (or a delegate) must use  
4242 TPM\_Delegate\_CreateOwnerDelegation with increment == TRUE. That blob permits creation  
4243 of new delegations or approval of existing tables and blobs. That delegation must set the  
4244 PCRs of the desired (local) process and the desired AuthData value of the process. As noted  
4245 previously, AuthData values should be a fixed value if the gathering entity is a trusted  
4246 process that is part of the normal software environment.

4247 If new delegations are to be created, TPM\_Delegate\_CreateOwnerDelegation must be used  
4248 with increment == FALSE.

4249 If existing blobs and delegation rows are to be reapproved,  
4250 TPM\_Delegate\_UpdateVerification must be used to install the new value of verificationCount  
4251 into those existing blobs and non-volatile rows. This exposes the blobs’ public information  
4252 (including the permissions and PCR values) in plain text to the transport session.

4253 TPM\_Delegate\_ReadTable then exposes all public values (including the permissions and  
4254 PCR values) of tables to the transport session.

4255 Again, after verifying all blobs, TPM\_ReleaseTransportSigned signs the list of transactions.

4256 **End of informative comment**

## 4257 **29.7 Use of commands for different states of TPM**

4258 **Start of informative comment**

4259 Use the ordinal table to determine when the various commands are available for use

4260 **End of informative comment**

## 4261 **29.8 Delegation Authorization Values**

4262 **Start of informative comment**

4263 This section describes why, when a PCR selection is set, the AuthData value may be a fixed  
4264 value, and, when the PCR selection is null, the delegation creator must select an AuthData  
4265 value.

4266 A PCR value is an indication of a particular (software) environment in the local platform.  
4267 Either that PCR value indicates a trusted process or not. If the trusted process is to execute  
4268 automatically, there is no point in allocating a meaningful AuthData value. (The only way  
4269 the trusted process could store the AuthData value is to seal it to the process’s PCR values,  
4270 but the delegation mechanism is already checking the process’s PCR values.) If execution of  
4271 the trusted process is dependent upon the wishes of another entity (such as the Owner), the  
4272 AuthData value should be a meaningful (private) value known only to the TPM, the Owner,  
4273 and that other entity. Otherwise the AuthData value should be a fixed, well known, value.

4274 If the delegation is to be controlled from a remote platform, these simple delegation  
4275 mechanisms provide no means for the platform to verify the PCRs of that remote platform,

4276and hence access to the delegation must be based solely upon knowledge of the AuthData  
4277value.

4278**End of informative comment**

## 4279**29.8.1 Using the authorization value**

4280**Start of informative comment**

4281To use a delegation the TPM will enforce any PCR selection on use. The use definition is any  
4282command that uses the delegation authorization value to take the place of the TPM Owner  
4283authorization.

4284**PCR Selection defined**

4285In this case, the delegation has a PCR selection structure defined. Each time the TPM uses  
4286the delegation authorization value instead of the TPM Owner value the TPM would validate  
4287that the current PCR settings match the settings held in the delegation structure. The PCR  
4288selection includes the definition of localities and checks of locality occur with the checking  
4289of the PCR values. The TPM enforces use of the correct authorization value, which may or  
4290may not be a meaningful (private) value.

4291**PCR selection NULL**

4292In this case, the delegation has no PCR selection structure defined. The TPM does not  
4293enforce any particular environment before using the authorization value. Mere knowledge of  
4294the value is sufficient.

4295**End of informative comment**

## 4296**29.9 DSAP description**

4297**Start of informative comment**

4298The DSAP opens a deferred auth session, using either a TPM\_DELEGATE\_BLOB as input  
4299parameter or a reference to the TPM\_DELEGATE\_TABLE\_ROW, stored inside the TPM. The  
4300DSAP command creates an ephemeral secret to authenticate a session. The purpose of this  
4301section is to illustrate the delegation of user keys or TPM Owner authorization by creating  
4302and using a DSAP session without regard to a specific command.

4303A key defined for a certain usage (e.g. TPM\_KEY\_IDENTITY) can be applied to different  
4304functions within the use model (e.g. TPM\_Quote or TPM\_CertifyKey). If an entity knows the  
4305AuthData for the key (key.usageAuth) it can perform all the functions, allowed for that use  
4306model of that particular key. This entity is also defined as delegation creation entity, since it  
4307can initiate the delegation process. Assume that a restricted usage entity should only be  
4308allowed to execute a subset or a single functions denoted as TPM\_Example, within the  
4309specific use model of a key. (e.g. Allow the usage of a TPM\_IDENTITY\_KEY only for  
4310Certifying Keys, but no other function). This use model points to the selection of the DSAP  
4311as the authorization protocol to execute the TPM\_Example command.

4312To perform this scenario the delegation creation entity must know the AuthData for the key  
4313(key.usageAuth). It then has to initiate the delegation by creating a  
4314TPM\_DELEGATE\_KEY\_BLOB via the TPM\_Delegate\_CreateKeyDelegation command. As a  
4315next step the delegation creation entity has to pass the TPM\_DELEGATE\_KEY\_BLOB and  
4316the delegation AuthData (TPM\_DELEGATE\_SENSITIVE.authValue) to the restricted usage

4317 entity. The specification offers the TPM\_DelTable\_ReadAuth mechanism to perform this  
4318 function. Other mechanisms may be used.

4319 The restricted usage entity can now start an TPM\_DSAP session by using the  
4320 TPM\_DELEGATE\_KEY\_BLOB as input.

4321 For the TPM\_Example command, the inAuth parameter provides the authorization to  
4322 execute the command. The following table shows the commands executed, the parameters  
4323 created and the wire formats of all of the information.

4324 <inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne,  
4325 inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode,  
4326 ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order:  
4327 authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the  
4328 following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

4329 In addition to the two even nonces generated by the TPM (authLastNonceEven and  
4330 nonceEven) that are used for TPM\_OIAP, there is a third, labeled nonceEvenOSAP that is  
4331 used to generate the shared secret. For every even nonce, there is also an odd nonce  
4332 generated by the system.

Caller	On the wire	Dir	TPM
Send TPM_DSAP	TPM_DSAP keyHandle nonceOddOSAP entityType entityValue	→	Decrypt sensitiveArea of entityValue If entityValue==TPM_ET_DEL_BLOB verify the integrity of the blob, and if a TPM_DELEGATE_KEY_BLOB is input verify that KeyHandle and entityValue match Create session & authHandle Generate authLastNonceEven Save authLastNonceEven with authHandle Generate nonceEvenOSAP Generate sharedSecret = HMAC(sensitiveArea.authValue., nonceEvenOSAP, nonceOddOSAP) Save keyHandle, sharedSecret with authHandle and permissions
Save authHandle, authLastNonceEven Generate sharedSecret = HMAC(sensitiveArea.authValue, nonceEvenOSAP, nonceOddOSAP) Save sharedSecret	authHandle, authLastNonceEven nonceEvenOSAP	←	Returns
Generate nonceOdd & save with authHandle. Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)			
Send TPM_Example	tag paramSize ordinal inArgOne inArgTwo authHandle nonceOdd continueAuthSession inAuth	→	Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Check if command ordinal of TPM_Example is allowed in permissions. If not return TPM_DISABLED_CMD Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

4335

4336 Suppose now that the TPM user wishes to send another command using the same session  
4337 to operate on the same key. For the purposes of this example, we will assume that the same  
4338 ordinal is to be used (TPM\_Example). To re-use the previous session, the  
4339 continueAuthSession output boolean must be TRUE.

4340 The following table shows the command execution, the parameters created and the wire  
4341 formats of all of the information.

4342 In this case, authLastNonceEven is the nonceEven value returned by the TPM with the  
4343 output parameters from the first execution of TPM\_Example.

Caller	On the wire	Dir	TPM
Generate nonceOdd Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Save nonceOdd with authHandle			
Send TPM_Example	tag paramSize ordinal inArgOne inArgTwo nonceOdd continueAuthSession inAuth	→	Retrieve authLastNonceEven from internal session storage HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL Execute TPM_Example and create returnCode Generate nonceEven to replace authLastNonceEven in session Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
Save nonceEven HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams) Compare HM to resAuth. This verifies returnCode and output parameters.	tag paramSize returnCode outArgOne nonceEven continueAuthSession resAuth	←	Return output parameters If continueAuthSession is FALSE then destroy session

4344

4345 The TPM user could then use the session for further authorization sessions or terminate it  
4346 in the ways that have been described above in TPM\_OIAP. Note that termination of the  
4347 DSAP session causes the TPM to destroy the shared secret.

#### 4348 **End of informative comment**

4349 1. The DSAP session MUST enforce any PCR selection on use. The use definition is any  
4350 command that uses the delegation authorization value to take the place of the TPM  
4351 Owner authorization.



## 4352**30. Physical Presence**

### 4353**Start of informative comment**

4354Physical presence is a signal from the platform to the TPM that indicates the operator  
4355manipulated the hardware of the platform. Manipulation would include depressing a  
4356switch, setting a jumper, depressing a key on the keyboard or some other such action.

4357TCG does not specify an implementation technique. The guideline is the physical presence  
4358technique should make it difficult or impossible for rogue software to assert the physical  
4359presence signal.

4360A PC-specific physical presence mechanism might be an electrical connection from a switch,  
4361or a program that loads during power on self-test.

### 4362**End of informative comment**

4363The TPM MUST support a signal from the platform for the assertion of physical presence. A  
4364TCG platform specific specification MAY specify what mechanisms assert the physical  
4365presence signal.

4366The platform manufacturer MUST provide for the physical presence assertion by some  
4367physical mechanism.

## 4368**30.1 Use of Physical Presence**

### 4369**Start of informative comment**

4370For control purposes there are numerous commands on the TPM that require TPM Owner  
4371authorization. Included in this group of commands are those that turn the TPM on or off  
4372and those that define the operating modes of the TPM. The TPM Owner always has complete  
4373control of the TPM. What happens in two conditions: there is no TPM Owner or the TPM  
4374Owner forgets the TPM Owner AuthData value. Physical presence allows for an  
4375authorization to change the state in these two conditions.

### 4376**No TPM Owner**

4377This state occurs when the TPM ships from manufacturing (it can occur at other times  
4378also). There is no TPM Owner. It is imperative to protect the TPM from remote software  
4379processes that would attempt to gain control of the TPM. To indicate to the TPM that the  
4380TPM operating state can change (allow for the creation of the TPM Owner) the human  
4381asserts physical presence. The physical presence assertion then indicates to the TPM that  
4382changing the operating state of the TPM is authorized.

### 4383**Lost TPM Owner authorization**

4384In the case of lost, or forgotten, authorization there is a TPM Owner but no way to manage  
4385the TPM. If the TPM will only operate with the TPM Owner authorization then the TPM is no  
4386longer controllable. Here the operator of the machine asserts physical presence and  
4387removes the current TPM Owner. The assumption is that the operator will then immediately  
4388take ownership of the TPM and insert a new TPM Owner AuthData value.

### 4389**Operator disabling**

4390Another use of physical presence is to indicate that the operator wants to disable the use of  
4391the TPM. This allows the operator to temporarily turn off the TPM but not change the  
4392permanent operating mode of the TPM as set by the TPM Owner.

4393**End of informative comment**

## 4394**31. TPM Internal Asymmetric Encryption**

### 4395**Start of Informative comment**

4396For asymmetric encryption schemes, the TPM is not required to perform the blocking of  
4397information where that information cannot be encrypted in a single cryptographic  
4398operation. The schemes TPM\_ES\_RSAESOAEP\_SHA1\_MGF1 and TPM\_ES\_RSAESPKCSV15  
4399allow only single block encryption. When using these schemes, the caller to the TPM must  
4400perform any blocking and unblocking outside the TPM. It is the responsibility of the caller  
4401to ensure that multiple blocks are properly protected using a chaining mechanism.

4402Note that there are inherent dangers associated with splitting information so that it can be  
4403encrypted in multiple blocks with an asymmetric key, and then chaining together these  
4404blocks together. For example, if an integrity check mechanism is not used, an attacker can  
4405encrypt his own data using the public key, and substitute this rogue block for one of the  
4406original blocks in the message, thus forcing the TPM to replace part of the message upon  
4407decryption.

4408There is also a more subtle attack to discover the data encrypted in low-entropy blocks. The  
4409attacker makes a guess at the plaintext data, encrypts it, and substitutes the encrypted  
4410guess for the original block. When the TPM decrypts the complete message, a successful  
4411decryption will indicate that his guess was correct.

4412There are a number of solutions which could be considered for this problem – One such  
4413solution for TPMs supporting symmetric encryption is specified in PKCS#7, section 10, and  
4414involves using the public key to encrypt a symmetric key, then using that symmetric key to  
4415encrypt the long message.

4416For TPMs without symmetric encryption capabilities, an alternative solution may be to add  
4417random padding to each message block, thus increasing the block's entropy.

4418This normative was deleted, since it contradicted Part 3: “For a TPM\_UNBIND command  
4419where the parent key has pubKey.algorithmId equal to TPM\_ALG\_RSA and  
4420pubKey.encScheme set to TPM\_ES\_RSAESPKCSv15 the TPM SHALL NOT expect a  
4421PAYLOAD\_TYPE structure to prepend the decrypted data.” The contradiction was the case  
4422of a TPM\_ES\_RSAESPKCSv15 binding key, which does have a payload.

### 4423**End of informative comment**

44241. The TPM MUST perform the encryption or decryption in accordance with the  
4425 specification of the encryption scheme, as described below.

44262. When a null terminated string is included in a calculation, the terminating null SHALL  
4427 NOT be included in the calculation.

### 4428**31.1.1 TPM\_ES\_RSAESOAEP\_SHA1\_MGF1**

44291. The encryption and decryption MUST be performed using the scheme RSA\_ES\_OAEP  
4430 defined in [PKCS #1v2.0: 7.1] using SHA1 as the hash algorithm for the encoding  
4431 operation.

44322. Encryption

4433 a. The OAEP encoding P parameter MUST be the 4 character string “TCPA”.

- 4434 b. While the TCG now controls this specification the string value will NOT change to  
4435 allow for interoperability and backward compatibility with TCG 1.1 TPM's
- 4436 c. If there is an error with the encryption, the TPM must return the error  
4437 TPM\_ENCRYPT\_ERROR.

#### 44383. Decryption

- 4439 a. The OAEP decoding P parameter MUST be the 4 character string "TCG".
- 4440 b. While the TCG now controls this specification the string value will NOT change to  
4441 allow for interoperability and backward compatibility with TCG 1.1 TPM's
- 4442 c. If there is an error with the decryption, the TPM must return the error  
4443 TPM\_DECRYPT\_ERROR.

### 444431.1.2 TPM\_ES\_RSAESPKCSV15

44451. The encryption MUST be performed using the scheme RSA\_ES\_PKCSV15 defined in  
4446 [PKCS #1v2.0: 7.2].

#### 44472. Encryption

- 4448 a. If there is an error with the encryption, return the error TPM\_ENCRYPT\_ERROR.

#### 44493. Decryption

- 4450 a. If there is an error with the decryption, return the error TPM\_DECRYPT\_ERROR.

### 445131.1.3 TPM\_ES\_SYM\_CTR

#### 4452Start of informative comment

4453This defines an encryption mode in use with symmetric algorithms. The actual definition is  
4454at

4455<http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>

4456The underlying symmetric algorithm may be AES128, AES192, or AES256. The definition  
4457for these algorithms is in the NIST document Appendix E.

4458The method of incrementing the counter value is different from that used by some standard  
4459crypto libraries (e.g. openssl, Java JCE) that increment the entire counter value. TPM  
4460users should be aware of this to avoid errors when the counter wraps.

#### 4461End of informative comment

44621. Given a current counter value, the next counter value is obtained by treating the lower  
4463 32 bits of the current counter value as an unsigned 32-bit integer  $x$ , then replacing the  
4464 lower 32 bits of the current counter value with the bits of the incremented integer  $(x + 1)$   
4465 mod  $2^{32}$ . This method is described in Appendix B.1 of the NIST document ( $m=32$ ).

### 446631.1.4 TPM\_ES\_SYM\_OFB

#### 4467Start of informative comment

4468This defines an encryption mode in use with symmetric algorithms. The actual definition is  
4469at

4470<http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>

4471The underlying symmetric algorithm may be AES128, AES192, or AES256. The definition  
4472for these algorithms is in the NIST document Appendix E.

4473**End of informative comment**

## 4474**31.2 TPM Internal Digital Signatures**

4475**Start of informative comment**

4476These values indicate the approved schemes in use by the TPM to generate digital  
4477signatures.

4478TPM 1.1 included only \_SHA1 keys. These allowed the TPM\_Sign command to sign a hash  
4479with no structure. This signature scheme is retained for backward compatibility.

4480TPM 1.2 added \_INFO keys to ensure that a structure, rather than a plain hash, is always  
4481signed. For TPM\_Sign, this signature scheme signs a new TPM\_SIGN\_INFO structure.  
4482Other ordinals, such as (e.g., TPM\_GetAuditDigestSigned, TPM\_CertifyKey, TPM\_Quote, etc.)  
4483inherently sign a structure, so the \_SHA1 and \_INFO signature schemes produce an  
4484identical result.

4485**End of informative comment**

4486The TPM MUST perform the signature or verification in accordance with the specification of  
4487the signature scheme, as described below.

### 4488**31.2.1 TPM\_SS\_RSASSAPKCS1v15\_SHA1**

4489**Start of informative comment**

4490This signature scheme prepends an OID to a SHA-1 digest. The OID, as specified in the  
4491normative, is as follows:

4492PKCS#1 v2.0: 8.1 says to encode the message per PKCS#1 v2.0: 9.2.1.

4493PKCS#1 v2.0: 9.2.1 says to apply the digest and then add the algorithm ID per Section 11.

4494PKCS#1 v2.0: Section 11.2.3 for SHA-1 says

4495 {iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26 }

4496and also

4497 For each OID, the parameters field associated with this OID in an AlgorithmIdentifier  
4498shall have type NULL.

4499The DER/BER Guide says that the first sub-identifiers are coded as  $40 * \text{value1} + \text{value2}$ .

4500Thus, the OID becomes (with comments):

45010x30 SEQUENCE

45020x21 33 bytes

4503 0x30 SEQUENCE

4504 0x09 9 bytes

4505 0x06 OID

4506 0x05 5 bytes

4507 0x2b 43 =  $40 * 1 (\text{iso}) + 3 (\text{identified-organization})$

4508 0x0e 14 from 11.2.3

4509 0x03 3 from 11.2.3

4510 0x02 2 from 11.2.3  
4511 0x1a 26 from 11.2.3  
4512 0x05 NULL (parameters)  
4513 0x00 0 bytes  
4514 0x04 OCTET  
4515 0x14 20 bytes (the SHA-1 digest to follow)

4516 **End of informative comment**

4517 1. The signature MUST be performed using the scheme RSASSA-PKCS1-v1.5 defined in  
4518 [PKCS #1v2.0: 8.1] using SHA1 as the hash algorithm for the encoding operation.

## 4519 **31.2.2 TPM\_SS\_RSASSAPKCS1v15\_DER**

4520 **Start of informative comment**

4521 This signature scheme is designed to permit inclusion of DER coded information before  
4522 signing, which is inappropriate for most TPM capabilities

4523 **End of informative comment**

4524 1. The signature MUST be performed using the scheme RSASSA-PKCS1-v1.5 defined in  
4525 [PKCS #1v2.0: 8.1]. The caller must properly format the area to sign using the DER  
4526 rules. The provided area maximum size is k-11 octets.

4527 2. TPM\_Sign SHALL be the only TPM capability that is permitted to use this signature  
4528 scheme. If a capability other than TPM\_Sign is requested to use this signature scheme,  
4529 it SHALL fail with the error code TPM\_INAPPROPRIATE\_SIG

## 4530 **31.2.3 TPM\_SS\_RSASSAPKCS1v15\_INFO**

4531 **Start of informative comment**

4532 This signature scheme is designed to permit signatures on arbitrary information but also  
4533 protect the signature mechanism from being misused.

4534 **End of informative comment**

4535 1. The scheme MUST work just as TPM\_SS\_RSASSAPKCS1v15\_SHA1 except in the  
4536 TPM\_Sign command

4537 a. In the TPM\_Sign command the scheme MUST use a properly constructed  
4538 TPM\_SIGN\_INFO structure, and hash it before signing

## 4539 **31.2.4 Use of Signature Schemes**

4540 **Start of informative comment**

4541 The TPM\_SS\_RSASSAPKCS1v15\_INFO scheme is a new addition for 1.2. It causes a new  
4542 functioning for 1.1 and 1.2 keys. The following details the use of the new scheme and how  
4543 the TPM handles signatures and hashing

4544 **End of informative comment**

4545 1. For commands that sign a TPM\_SIGN\_INFO structure (e.g.,  
4546 (TPM\_GetAuditDigestSigned, TPM\_TickStampBlob, TPM\_ReleaseTransportSigned)

713

- 4547 a. The TPM MUST create a TPM\_SIGN\_INFO and sign using the  
4548 TPM\_SS\_RSASSAPKCS1v15\_SHA1 scheme for either \_SHA1 or \_INFO keys.
45492. For commands that sign a structure defined by the command (e.g.,  
4550 (TPM\_CMK\_CreateTicket, TPM\_CertifyKey, TPM\_CertifyKey2, TPM\_MakeIdentity,  
4551 TPM\_Quote, TPM\_Quote2, TPM\_CertifySelfTest, TPM\_GetCapabilitySigned)
- 4552 a. Create the structure as defined by the command and sign using the  
4553 TPM\_SS\_RSASSAPKCS1v15\_SHA1 scheme for either \_SHA1 or \_INFO keys.
45543. For TPM\_Sign:
- 4555 a. Create the structure as defined by the command and key scheme
- 4556 b. If key->sigScheme is TPM\_SS\_RSASSAPKCS1v15\_SHA1, sign the 20 byte parameter
- 4557 c. If key->sigScheme is TPM\_SS\_RSASSAPKCS1v15\_DER, sign the DER value.
- 4558 d. If key->sigScheme is TPM\_SS\_RSASSAPKCS1v15\_INFO, sign any value using the  
4559 TPM\_SIGN\_INFO structure.
45604. When data is signed and the data comes from INSIDE the TPM, the TPM MUST do the  
4561 hash, and prepend the DER encoding correctly before performing the padding and  
4562 private key operation.
45635. When data is signed and the data comes from OUTSIDE the TPM, the software, not the  
4564 TPM, MUST do the hash.
45656. When the TPM knows, or is told by implication, that the hash used is SHA-1, the TPM  
4566 MUST prepend the DER encoding correctly before performing the padding and private  
4567 key operation
45687. When the TPM does not know, or told by implication, that the hash used is SHA-1, the  
4569 software, not the TPM) MUST provide the DER encoding to be prepended.
45708. The TPM MUST perform the padding and private key operation in any signing operations  
4571 it does.

## 32. Key Usage Table

### Start of informative comment

Asymmetric keys (e.g., RSA keys) can do two basic functions: sign/verify and encrypt/decrypt.

TPM\_KEY\_SIGNING and TPM\_KEY\_IDENTITY do signature functions.

TPM\_KEY\_STORAGE, TPM\_KEY\_BIND, TPM\_KEY\_MIGRATE, and TPM\_KEY\_AUTHCHANGE do encryption functions.

### End of informative comment

This table summarizes the types of keys associated with a given TPM command.

It is the responsibility of each command to check the key usage prior to executing the command

Name	First Key	Second Key	First Key						Second Key					
			SIGNING	STORAGE	IDENTITY	AUTHCHG	BIND	LEGACY	SIGNING	STORAGE	IDENTITY	AUTHCHG	BIND	LEGACY
TPM_ActivateIdentity	idKey				x									
TPM_CertifyKey	certKey	inKey	x		x			x	x	x	x		x	x
TPM_CertifyKey2 (Note 3)	inKey	certKey	x	x	x		x	x	x		x			x
TPM_CertifySelfTest	key		x		x			x						
TPM_ChangeAuth	parent	blob		x					2	2	2	2	2	2
TPM_ChangeAuthAsymFinish	parent	ephemeral		x								x		
TPM_ChangeAuthAsymStart	idKey	ephemeral			x							x		
TPM_CMK_ConvertMigration	parent			x										
TPM_CMK_CreateBlob	parent			x										
TPM_CMK_CreateKey	parent			x										
TPM_ConvertMigrationBlob	parent			x										
TPM_CreateMigrationBlob	parent	blob		x					2	2	2	2	2	2
TPM_CreateWrapKey	parent			x										
TPM_Delegate_CreateKeyDelegation	key		x	x	x	x	x	x						
TPM_DSAP	entity		x	x	x	x	x	x						
TPM_EstablishTransport	key			x				x						
TPM_GetAuditDigestSigned	certKey		x		x			x						
TPM_GetAuditEventSigned	certKey		x					x						
TPM_GetCapabilitySigned	key		x		x			x						



TPM_GetPubKey	key		x	x	x	x	x	x											
TPM_KeyControlOwner	key		x	x	x			x	x										
TPM_LoadKey2	parent	inKey		x						x	x	x			x	x			
TPM_LoadKey	parent	inKey		x						x	x	x			x	x			
TPM_MigrateKey	maKey			1															
TPM_OSAP	entity		x	x	x	x	x	x											
TPM_Quote	key		x		x					x									
TPM_Quote2	key		x		x					x									
TPM_Seal	key			x															
TPM_Sealx	key			x															
TPM_Sign	key		x							x									
TPM_UnBind	key								x	x									
TPM_Unseal	parent			x															
TPM_ReleaseTransportSigned	key		x																
TPM_TickStampBlob	key		x		x					x									

4583**Notes**

- 45841 – Key is not a storage key but TPM\_MIGRATE\_KEY
- 45852 – TPM unable to determine key type
- 45863 – The order is correct; the reason is to support a single auth version.

## 4587 **33. Direct Anonymous Attestation**

### 4588 **Start of informative comment**

4589 TPM\_DAA\_Join and TPM\_DAA\_Sign are highly resource intensive commands. They require  
4590 most of the internal TPM resources to accomplish the complete set of operations. A TPM  
4591 may specify that no other commands are possible during the join or sign operations. To  
4592 allow other operations to occur, the TPM does allow the TPM\_SaveContext command to save  
4593 off the current join or sign operation.

4594 Operations that occur during a join or sign result in the loss of the join or sign session in  
4595 favor of the interrupting command.

### 4596 **End of informative comment**

4597 1. The TPM MUST support one concurrent TPM\_DAA\_Join or TPM\_DAA\_Sign session. The  
4598 TPM MAY support additional sessions

4599 2. The TPM MAY invalidate a join or sign session upon the receipt of any additional  
4600 command other than the join/sign or TPM\_SaveContext

## 4601 **33.1 TPM\_DAA\_JOIN**

### 4602 **Start of informative comment**

4603 TPM\_DAA\_Join creates new JOIN data. If a TPM supports only one JOIN/SIGN operation,  
4604 TPM\_DAA\_Join invalidates any previous DAA attestation information inside a TPM. The  
4605 JOIN phase of a DAA context requires a TPM to communicate with an issuer.  
4606 TPM\_DAA\_Join outputs data to be sent to an issuing authority and receives data from that  
4607 issuing authority. The operation potentially requires several seconds to complete, but is  
4608 done in a series of atomic stages and TPM\_SaveContext/TPM\_LoadContext can be used to  
4609 cache data off-TPM in between atomic stages.

4610 The JOIN process is designed so a TPM will normally receive exactly the same DAA  
4611 credentials from a given issuer, no matter how many times the JOIN process is executed  
4612 and no matter whether the issuer changes his keys. This property is necessary because an  
4613 issuer must give DAA credentials to a platform after verifying that the platform has the  
4614 architecture of a trusted platform. Unless the issuer repeats the verification process, there  
4615 is no justification for giving different DAA credentials to the same platform. Even after  
4616 repeating the verification process, the issuer should give replacement (different) DAA  
4617 credentials only when it is necessary to retire the old DAA credentials. Replacement DAA  
4618 credentials erase the previous DAA history of the platform, at least as far as the DAA  
4619 credentials from that issuer are concerned. Replacement might be desirable, as when a  
4620 platform changes hands, for example, in order to eliminate any association via DAA between  
4621 the seller and the buyer. On the other hand, replacement might be undesirable, since it  
4622 enables a rogue to rejoin a community from which he has been barred. Replacement is done  
4623 by submitting a different “count” value to the TPM during a JOIN process. A platform may  
4624 use any value of “count” at any time, in any order, but only “counts” accepted by the issuer  
4625 will elicit DAA credentials from that issuer.

4626 The TPM is forced to verify an issuer’s public parameters before using an issuer’s public  
4627 parameters. This verification provides proof that the public parameters (which include a  
4628 public key) were approved by an entity that knows the private key corresponding to that  
4629 public key; in other words that the JOIN has previously been approved by the issuer. This

4630verification is necessary to prevent an attack by a rogue using a genuine issuer's public  
4631parameters, which could reveal the secret created by the TPM using those public  
4632parameters. Verification uses a signature (provided by the issuer) over the public  
4633parameters.

4634The exponent of the issuer's key is fixed at  $2^{16}+1$ , because this is the only size of exponent  
4635that a TPM is required to support. The modulus of the issuer's public key is used to create  
4636the pseudonym with which the TPM contacts the issuer. Hence, the TPM cannot produce  
4637the same pseudonym for different issuers (who have different keys). The pseudonym is  
4638always created using the issuer's first key, even if the issuer changes keys, in order to  
4639produce the property described earlier. The issuer proves to the TPM that he has the right  
4640to use that first key to create a pseudonym by creating a chain of signatures from the first  
4641key to the current key, and submitting those signatures to the TPM. The method has the  
4642desirable property that only signatures and the most recent private key need be retained by  
4643the issuer: once the latest link in the signature chain has been created, previous private  
4644keys can be discarded.

4645The use of atomic operations minimizes the contiguous time that a TPM is busy with  
4646TPM\_DAA\_Join and hence unavailable for other commands. JOIN can therefore be done as  
4647a background activity without inconveniencing a user. The use of atomic operations also  
4648minimizes the peak value of TPM resources consumed by the JOIN phase.

4649The use of atomic operations introduces a need for consistency checks, to ensure that the  
4650same parameters are used in all atomic operations of the same JOIN process.  
4651DAA\_tpmSpecific therefore contains a digest of the associated DAA\_issuerSettings  
4652structure, and DAA\_session contains a digest of associated DAA\_tpmSpecific and  
4653DAA\_joinSession structures. Each atomic operation verifies digests to ensure use of  
4654mutually consistent sets of DAA\_issuerSettings, DAA\_tpmSpecific, DAA\_session, and  
4655DAA\_joinSession data.

4656JOIN operations and data structures are designed to minimize the amount of data that  
4657must be stored on a TPM in between atomic operations, while ensuring use of mutually  
4658consistent sets of data. Digests of public data are held in the TPM between atomic  
4659operations, instead of the actual public data (if a digest is smaller than the actual data). In  
4660each atomic operation, consistency checks verify that any public data loaded and used in  
4661that operation matches the stored digest. Thus non-secret DAA\_generic\_X parameters  
4662(loaded into the TPM only when required), are checked using digests DAA\_digest\_X  
4663(preloaded into the TPM in the structure DAA\_issuerSettings).

4664JOIN includes a challenge from the issuer, in order to defeat simple Denial of Service  
4665attacks on the issuer's server by rogues pretending to be arbitrary TPMs.

4666A first group of atomic operations generate all TPM-data that must be sent to the issuer.  
4667The platform performs other operations (that do not need to be trusted) using the TPM-data,  
4668and sends the resultant data to the issuer. The issuer sends values u2 and u3 back to the  
4669TPM. A second group of atomic operations accepts this data from the issuer and completes  
4670the protocol.

4671The TPM outputs encrypted forms of DAA\_tpmSpecific, v0 and v1. These encrypted data are  
4672later interpreted by the same TPM and not by any other entity, so any manufacturer-  
4673specific wrapping can be used. It is suggested, however, that enc(DAA\_tpmSpecific) or  
4674enc(v0) or enc(v1) data should be created by adapting a TPM\_CONTEXT\_BLOB structure.

4675 After executing TPM\_DAA\_Join, it is prudent to perform TPM\_DAA\_Sign, to verify that the  
4676 JOIN process completed correctly. A host platform may choose to verify JOIN by performing  
4677 TPM\_DAA\_Sign as both the target and the verifier (or could, of course, use an external  
4678 verifier).

4679 **End of informative comment**

## 4680 **33.2 TPM\_DAA\_Sign**

4681 **Start of informative comment**

4682 TPM\_DAA\_Sign responds to a challenge and proves the attestation held by a TPM without  
4683 revealing the attestation held by that TPM. The operation is done in a series of atomic  
4684 stages to minimize the contiguous time that a TPM is busy and hence unavailable for other  
4685 commands. TPM\_SaveContext can be used to save a DAA context in between atomic stages.  
4686 This enables the response to the challenge to be done as a background activity without  
4687 inconveniencing a user, and also minimizes the peak value of TPM resources consumed by  
4688 the process.

4689 The use of atomic operations introduces a need for consistency checks, to ensure that the  
4690 same parameters are used in all atomic operations of the same SIGN process.  
4691 DAA\_tpmSpecific therefore contains a digest of the associated DAA\_issuerSettings  
4692 structure, and DAA\_session contains a digest of associated DAA\_tpmSpecific structure.  
4693 Each atomic operation verifies these digests and hence ensures use of mutually consistent  
4694 sets of DAA\_issuerSettings, DAA\_tpmSpecific, and DAA\_session data.

4695 SIGN operations and data structures are designed to minimise the amount of data that  
4696 must be stored on a TPM in between atomic operations, while ensuring use of mutually  
4697 consistent sets of data. Digests of public and private data are held in the TPM between  
4698 atomic operations, instead of the actual public or private data (if a digest is smaller than the  
4699 actual data). At each atomic operation, consistency checks verify that any data loaded and  
4700 used in that operation matches the stored digest. Thus parameters DAA\_digest\_X are  
4701 digests (preloaded into the TPM in the structure DAA\_issuerSettings) of non-secret  
4702 DAA\_generic\_X parameters (loaded into the TPM only when required), for example.

4703 The design enables the use of any number of issuer DAA-data, private DAA-data, and so on.  
4704 Strictly, the design is that the \*TPM\* puts no limit on the number of sets of issuer DAA-data  
4705 or sets of private DAA-data, or restricts what set is in the TPM at any time, but supports  
4706 only one DAA-context in the TPM at any instant. Any number of DAA-contexts can, of  
4707 course, be swapped in and out of the TPM using TPM\_SaveContext/TPM\_LoadContext, so  
4708 applications do not perceive a limit on the number of DAA contexts.

4709 TPM\_DAA\_Sign accepts a freshness challenge from the verifier and generates all TPM-data  
4710 that must be sent to the verifier. The platform performs other operations (that do not need  
4711 to be trusted) using the TPM-data, and sends the resultant data to the verifier. At one stage,  
4712 the TPM incorporates a loaded public (non-migratable) key into the protocol. This is  
4713 intended to permit the setup of a session, for any specific purpose, including doing the  
4714 same job in TPM\_ActivateIdentity as the EK.

4715 **End of informative comment**

## 4716 **33.3 DAA Command summary**

4717 **Start of informative comment**

4718The following is a conceptual summary of the operations that are necessary to setup a TPM  
4719for DAA, execute the JOIN process, and execute the SIGN process.

4720The summary is partitioned according to the “stages” of the actual TPM commands. Thus,  
4721the operations listed in JOIN under stage-2 briefly describe the operation of TPM\_DAA\_Join  
4722at stage-2, for example.

4723This summary is in place to help in the connection between the mathematical definition of  
4724DAA and this implementation in a TPM.

4725**End of informative comment**

### 4726**33.3.1 TPM setup**

47271. A TPM generates a TPM-specific secret  $S$  (160-bit) from the RNG and stores  $S$  in  
4728 nonvolatile store on the TPM. This value will never be disclosed and changed by the  
4729 TPM.

### 4730**33.3.2 JOIN**

4731**Start of informative comment**

4732This entire section is informative

47331. When the following is performed, this process does not increment the stage counter.

4734a. TPM imports a non-secret values  $n_0$  (2048-bit).

4735b. TPM computes a non-secret value  $N_0$  (160-bit) =  $H(n_0)$ .

4736c. TPM computes a TPM-specific secret  $DAA\_rekey$  (160-bit) =  $H(S, H(n_0))$ .

4737d. TPM stores a self-consistent set of ( $N_0$ ,  $DAA\_rekey$ )

47382. The following is performed 0 or several times: (Note: If the stage mechanism is being  
4739used, then this branch does not increment the stage counter.)

4740a. TPM imports

4741i. a self consistent set of ( $N_0$ ,  $DAA\_rekey$ )

4742ii. a non-secret value  $DAA\_SEED\_KEY$  (2048-bit)

4743iii. a non-secret value  $DEPENDENT\_SEED\_KEY$  (2048-bit)

4744iv. a non-secret value  $SIG\_DSK$  (2048-bit)

4745b. TPM computes  $DIGEST$  (160-bit) =  $H(DAA\_SEED\_KEY)$

4746c. If  $DIGEST \neq N_0$ , TPM refuses to continue

4747d. If  $DIGEST == N_0$ , TPM verifies validity of signature  $SIG\_DSK$  on  
4748  $DEPENDENT\_SEED\_KEY$  with key ( $DAA\_SEED\_KEY$ ,  $e_0 (= 2^{16} + 1)$ ) by using  
4749  $TPM\_Sign\_Verify$  (based on PKCS#1 2.0). If check fails, TPM refuses to continue.

4750e. TPM sets  $N_0 = H(DEPENDENT\_SEED\_KEY)$

4751f. TPM stores a self consistent set of ( $N_0$ ,  $DAA\_JOIN$ )

47523. Stage 2

4753a. TPM imports a set of values, including

4754i. a non-secret value  $n_0$  (2048-bit),  
4755ii. a non-secret value  $R_0$  (2048-bit),  
4756iii. a non-secret value  $R_1$  (2048-bit),  
4757iv. a non-secret value  $S_0$  (2048-bit),  
4758v. a non-secret value  $S_1$  (2048-bit),  
4759vi. a non-secret value  $n$  (2048-bit),  
4760vii. a non-secret value  $n_1$  (1024-bit),  
4761viii. a non-secret value  $\gamma$  (2048-bit),  
4762ix. a non-secret value  $q$  (208-bit),  
4763x. a non-secret value COUNT (8-bit),  
4764xi. a self consistent set of ( $N_0$ , DAA\_rekey).  
4765xii. TPM saves them as part of a new set A.  
4766b. TPM computes DIGEST (160-bit) =  $H(n_0)$   
4767c. If DIGEST  $\neq N_0$ , TPM refuses to continue.  
4768d. If DIGEST  $= N_0$ , TPM computes DIGEST (160-bit) =  $H(R_0, R_1, S_0, S_1, n, n_1, \Gamma, q)$   
4769e. TPM imports a non-secret value SIG\_ISSUER\_KEY (2048-bit).  
4770f. TPM verifies validity of signature SIG\_ISSUER\_KEY (2048-bit) on DIGEST with key ( $n_0$ ,  
4771e0) by using TPM\_Sign\_Verify (based on PKCS#1 2.0). If check fails, TPM refuses to  
4772continue.  
4773g. TPM computes a TPM-specific secret  $f$  (208-bit) =  $H(\text{DAA\_rekey}, \text{COUNT}, 0) ||$   
4774 $H(\text{DAA\_rekey}, \text{COUNT}, 1) \bmod q$ .  
4775h. TPM computes a TPM-specific secret  $f_0$  (104-bit) =  $f \bmod 2^{104}$ .  
4776i. TPM computes a TPM-specific secret  $f_1$  (104-bit) =  $f \gg 104$ .  
4777j. TPM save  $f$ ,  $f_0$  and  $f_1$  as part of set A.  
47784. Stage 3  
4779a. TPM generates a TPM-specific secret  $u_0$  (1024-bit) from the RNG.  
4780b. TPM generates a TPM-specific secret  $u'_1$  (1104-bit) from the RNG.  
4781c. TPM computes  $u_1$  (1024-bit) =  $u'_1 \bmod n_1$ .  
4782d. TPM stores  $u_0$  and  $u_1$  as part of set A.  
47835. Stage 4  
4784a. TPM computes a non-secret value  $P_1$  (2048-bit) =  $(R_0^{f_0}) \bmod n$  and stores  $P_1$  as part of  
4785set A.  
47866. Stage 5  
4787a. TPM computes a non-secret value  $P_2$  (2048-bit) =  $P_1 * (R_1^{f_1}) \bmod n$ , stores  $P_2$  as part of  
4788set A and erases  $P_1$  from set A.  
47897. Stage 6

749

4790a. TPM computes a non-secret value  $P3$  (2048-bit) =  $P2 \cdot (S0^{u0}) \bmod n$ , stores  $P3$  as part of  
4791set A and erases  $P2$  from set A.

47928. Stage 7

4793a. TPM computes a non-secret value  $U$  (2048-bit) =  $P3 \cdot (S1^{u1}) \bmod n$ .

4794b. TPM erases  $P3$  from set A

4795c. TPM computes and saves  $U1$  (160-bit) =  $H(U \parallel \text{COUNT} \parallel N0)$  as part of set A.

4796d. TPM exports  $U$ .

47979. Stage 8

4798a. TPM imports ENC\_NE (2048-bit).

4799b. TPM decrypts NE (160-bit) from ENC\_NE (2048-bit) by using privEK: NE =  
4800decrypt(privEK, ENC\_NE).

4801c. TPM computes  $U2$  (160-bit) =  $H(U1 \parallel NE)$ .

4802d. TPM erases  $U1$  from set A.

4803e. TPM exports  $U2$ .

480410. Stage 9

4805a. TPM generates a TPM-specific secret  $r0$  (344-bit) from the RNG.

4806b. TPM generates a TPM-specific secret  $r1$  (344-bit) from the RNG.

4807c. TPM generates a TPM-specific secret  $r2$  (1024-bit) from the RNG.

4808d. TPM generates a TPM-specific secret  $r3$  (1264-bit) from the RNG.

4809e. TPM stores  $r0$ ,  $r1$ ,  $r2$ ,  $r3$  as part of set A.

4810f. TPM computes a non-secret value  $P1$  (2048-bit) =  $(R0^{r0}) \bmod n$  and stores  $P1$  as part of  
4811set A.

481211. Stage 10

4813a. TPM computes a non-secret value  $P2$  (2048-bit) =  $P1 \cdot (R1^{r1}) \bmod n$ , stores  $P2$  as part of  
4814set A and erases  $P1$  from set A.

481512. Stage 11

4816a. TPM computes a non-secret value  $P3$  (2048-bit) =  $P2 \cdot (S0^{r2}) \bmod n$ , stores  $P3$  as part of  
4817set A and erases  $P2$  from set A.

481813. Stage 12

4819a. TPM computes a non-secret value  $P4$  (2048-bit) =  $P3 \cdot (S1^{r3}) \bmod n$ , stores  $P4$  as part of  
4820set A and erases  $P3$  from set A.

4821b. TPM exports  $P4$ .

482214. Stage 13

4823a. TPM imports  $w$  (2048-bit).

4824b. TPM computes  $w1 = w^q \bmod \Gamma$ .

4825c. TPM verifies if  $w1 = 1$  holds. If it doesn't hold, TPM refuses to continue.

4826d. If it does hold, TPM saves  $w$  as part of set  $A$ .

482715. Stage 14

4828a. TPM computes a non-secret value  $E$  (2048-bit) =  $w^f \bmod \Gamma$ .

4829b. TPM exports  $E$ .

483016. Stage 15

4831a. TPM computes a TPM-specific secret  $r$  (208-bit) =  $r_0 + 2^{104} \cdot r_1 \bmod q$ .

4832b. TPM computes a non-secret value  $E_1$  (2048-bit) =  $w^r \bmod \Gamma$ .

4833c. TPM exports  $E_1$  and erases  $w$  from set  $A$ .

483417. Stage 16

4835a. TPM imports a non-secret value  $c_1$  (160-bit).

4836b. TPM generates a non-secret value  $NT$  (160-bit) from the RNG.

4837c. TPM computes a non-secret value  $c$  (160-bit) =  $H(c_1 || NT)$ .

4838d. TPM save  $c$  as part of set  $A$ .

4839e. TPM exports  $NT$

484018. Stage 17

4841a. TPM computes a non-secret value  $s_0$  (352-bit) =  $r_0 + c \cdot f_0$  over the integers.

4842b. TPM exports  $s_0$ .

484319. Stage 18

4844a. TPM computes a non-secret value  $s_1$  (352-bit) =  $r_1 + c \cdot f_1$  over the integers.

4845b. TPM exports  $s_1$ .

484620. Stage 19

4847a. TPM computes a non-secret value  $s_2$  (1024-bit) =  $r_2 + c \cdot u_0 \bmod 2^{1024}$ .

4848b. TPM exports  $s_2$ .

484921. Stage 20

4850a. TPM computes a non-secret value  $s'_2$  (1024-bit) =  $(r_2 + c \cdot u_0) \gg 1024$  over the integers.

4851b. TPM saves  $s'_2$  as part of set  $A$ .

4852c. TPM exports  $c$

485322. Stage 21

4854a. TPM computes a non-secret value  $s_3$  (1272-bit) =  $r_3 + c \cdot u_1 + s'_2$  over the integers.

4855b. TPM exports  $s_3$  and erases  $s'_2$  from set  $A$ .

485623. Stage 22

4857a. TPM imports a non-secret value  $u_2$  (1024-bit).

4858b. TPM computes a TPM-specific secret  $v_0$  (1024-bit) =  $u_2 + u_0 \bmod 2^{1024}$ .

4859c. TPM stores  $v_0$  as part of  $A$ .



4860d. TPM computes a TPM-specific secret  $v_0$  (1024-bit) =  $(u_2 + u_0) \gg 1024$  over the integers.

4861e. TPM saves  $v_0$  as part of set A.

486224.Stage 23

4863a. TPM imports a non-secret value  $u_3$  (1512-bit).

4864b. TPM computes a TPM-specific secret  $v_1$  (1520-bit) =  $u_3 + u_1 + v_0$  over the integers.

4865c. TPM stores  $v_1$  as part of A.

4866d. TPM erases  $v_0$  from set A.

486725.Stage 24

4868a. TPM makes self-consistent set of all the data ( $n_0$ , COUNT,  $R_0$ ,  $R_1$ ,  $S_0$ ,  $S_1$ ,  $n$ ,  $\Gamma$ ,  $q$ ,  $v_0$ ,  $v_1$ ), where the values  $v_0$ ,  $v_1$  are secret – they need to be stored safely with the consistent set, and the remaining is non-secret.

4871b. TPM erases set A.

4872**End of informative comment**

### 4873**33.3.3 SIGN**

4874**Start of informative comment**

4875This entire section is informative

48761. Stage 0 & 1

4877a. TPM imports and verifies a self-consistent set of all the data including:

4878i. a non-secret value  $n_0$  (2048-bit),

4879ii. a non-secret value COUNT (8-bit),

4880iii. a non-secret value  $R_0$  (2048-bit),

4881iv. a non-secret value  $R_1$  (2048-bit),

4882v. a non-secret value  $S_0$  (2048-bit),

4883vi. a non-secret value  $S_1$  (2048-bit),

4884vii. a non-secret value  $n$  (2048-bit),

4885viii. a non-secret value  $\gamma$  (2048-bit),

4886ix. a non-secret value  $q$  (208-bit),

4887x.  $v_0$  (1024-bit),

4888xi.  $v_1$  (1520-bit).

4889xii. If the verification does not succeed, TPM refuses to continue.

4890b. TPM stores the above values as part of a new set A.

4891c. TPM computes a TPM-specific secret  $f_0$  (104-bit) =  $f \bmod 2^{104}$ .

4892d. TPM computes a TPM-specific secret  $f_1$  (104-bit) =  $f \gg 104$ .

4893e. TPM stores  $f_0$  and  $f_1$  as part of set A.

- 4894f. TPM generates a TPM-specific secret  $r_0$  (344-bit) from the RNG.
- 4895g. TPM generates a TPM-specific secret  $r_1$  (344-bit) from the RNG.
- 4896h. TPM generates a TPM-specific secret  $r_2$  (1024-bit) from the RNG.
- 4897i. TPM generates a TPM-specific secret  $r_4$  (1752-bit) from the RNG.
- 4898j. TPM stores  $r_0$ ,  $r_1$ ,  $r_2$ ,  $r_4$ , as part of set A.
- 48992. Stage 2
- 4900a. TPM computes a non-secret value  $P_1$  (2048-bit) =  $(R_0^{r_0}) \bmod n$  and stores  $P_1$  as part of set A.
- 4901set A.
- 49023. Stage 3
- 4903a. TPM computes a non-secret value  $P_2$  (2048-bit) =  $P_1 * (R_1^{r_1}) \bmod n$ , stores  $P_2$  as part of set A and erases  $P_1$  from set A.
- 4904set A
- 49054. Stage 4
- 4906a. TPM computes a non-secret value  $P_3$  (2048-bit) =  $P_2 * (S_0^{r_2}) \bmod n$ , stores  $P_3$  as part of set A and erases  $P_2$  from set A.
- 4907set A
- 49085. Stage 5
- 4909a. TPM computes a non-secret value  $T$  (2048-bit) =  $P_3 * (S_1^{r_4}) \bmod n$ .
- 4910b. TPM erases  $P_3$  from set A.
- 4911c. TPM exports  $T$ .
- 49126. Stage 6
- 4913a. TPM imports a non-secret value  $w$  (2048-bit).
- 4914b. TPM computes  $w_1 = w^q \bmod \Gamma$ .
- 4915c. TPM verifies if  $w_1 = 1$  holds. If it doesn't hold, TPM refuses to continue.
- 4916d. If it does hold, TPM saves  $w$  as part of set A.
- 49177. Stage 7
- 4918a. TPM computes a non-secret value  $E$  (2048-bit) =  $w^f \bmod \Gamma$ .
- 4919b. TPM exports  $E$  and erases  $f$  from set A.
- 49208. Stage 8
- 4921a. TPM computes a TPM-specific secret  $r$  (208-bit) =  $r_0 + 2^{104} * r_1 \bmod q$ .
- 4922b. TPM computes a non-secret value  $E_1$  (2048-bit) =  $w^r \bmod \Gamma$ .
- 4923c. TPM exports  $E_1$  and erases  $w$  and  $E_1$  from set A.
- 49249. Stage 9
- 4925a. TPM imports a non-secret value  $c_1$  (160-bit).
- 4926b. TPM generates a non-secret value  $NT$  (160-bit) from the RNG.
- 4927c. TPM computes a non-secret value  $c_2$  (160-bit) =  $H(c_1 || NT)$  and erases  $c_1$  from set A.
- 4928d. TPM saves  $c_2$  as part of set A.

4929e. TPM exports NT.

493010.Stage 10

4931a. TPM imports a non-secret value  $b$  (1-bit).

4932b. If  $b = 1$ , TPM imports a non-secret value  $m$  (160-bit).

4933c. TPM computes a non-secret value  $c$  (160-bit) =  $H(c2 || b || m)$  and erases  $c2$  from set A.

4934d. If  $b = 0$ , TPM imports an RSA public key,  $eAIK$  ( $= 2^{16} + 1$ ) and  $nAIK$  (2048-bit).

4935e. TPM computes a non-secret value  $c$  (160-bit) =  $H(c2 || b || nAIK)$  and erases  $c2$  from set A.

4937f. TPM exports  $c$ .

493811.Stage 11

4939a. TPM computes a non-secret value  $s0$  (352-bit) =  $r0 + c*f0$  over the integers.

4940b. TPM exports  $s0$ .

494112.Stage 12

4942a. TPM computes a non-secret value  $s1$  (352-bit) =  $r1 + c*f1$  over the integers.

4943b. TPM exports  $s1$ .

494413.Stage 13

4945a. TPM computes a non-secret value  $s2$  (1024-bit) =  $r2 + c*v0 \bmod 2^{1024}$ .

4946b. TPM exports  $s2$ .

494714.Stage 14

4948a. TPM computes a non-secret value  $s'2$  (1024-bit) =  $(r2 + c*v0) \gg 1024$  over the integers.

4949b. TPM saves  $s'2$  as part of set A.

495015.Stage 15

4951a. TPM computes a non-secret value  $s3$  (1760-bit) =  $r4 + cv1 + s'2$  over the integers.

4952b. TPM exports  $s3$  and erases  $s'2$  from set A.

4953c. TPM erases set A.

4954**End of informative comment**

## 4955 **34. General Purpose IO**

### 4956 **Start of informative comment**

4957 The GPIO capability allows an outside entity to output a signal on a GPIO pin, or read the  
4958 status of a GPIO pin. The solution is for a single pin, with no timing information. There is  
4959 no support for sending information on specific busses like SMBus or RS232. The design  
4960 does support the designation of more than one GPIO pin.

4961 There is no requirement as to the layout of the GPIO pin, or the routing of the wire from the  
4962 GPIO pin on the platform. A platform specific specification can add those requirements.

4963 To avoid the designation of additional command ordinals, the architecture uses the NV  
4964 Storage commands. A set of GPIO NV indexes map to individual GPIO pins.  
4965 TPM\_NV\_INDEX\_GPIO\_00 maps to the first GPIO pin. The platform specific specification  
4966 indicates the mapping of GPIO zero to a specific package pin.

4967 The TPM does not reserve any NV storage for the indicated pin; rather the TPM uses the  
4968 authorization mechanisms for NV storage to allow a rich set of controls on the use of the  
4969 GPIO pin. The TPM owner can specify when and how the platform can use the GPIO pin.  
4970 While there is no NV storage for the pin value, TRUE or FALSE, there is NV storage for the  
4971 authorization requirements for the pin.

4972 Using the NV attributes the GPIO pin may be either an input pin or an output pin.

### 4973 **End of informative comment**

4974 1. The TPM MAY support the use of a GPIO pin defined by the NV storage mechanisms.

4975 2. The GPIO pin MAY be either an input or an output pin.

## 4976**35. Redirection**

### 4977 **Informative comment**

4978Redirection allows the TPM to output the results of operations to hardware other than the  
4979normal TPM communication bus. The redirection can occur to areas internal or external to  
4980the TPM. Redirection is only available to key operations (such as TPM\_UnBind,  
4981TPM\_Unseal, and TPM\_GetPubKey). To use redirection the key must be created specifying  
4982redirection as one of the keys attributes.

4983When redirecting the output the TPM will not interpret any of the data and will pass the  
4984data on without any modifications.

4985The TPM\_SetRedirection command connects a destination location or port to a loaded key.  
4986This connection remains so long as the key is loaded, and is saved along with other key  
4987information on a saveContext(key), loadContext(key). If the key is reloaded using  
4988TPM\_LoadKey, then TPM\_SetRedirection must be run again.

4989Any use of TPM\_SetRedirection with a key that does not have the redirect attribute must  
4990return an error. Use of key that has the redirect attribute without TPM\_SetRedirection being  
4991set must return an error.

### 4992**End of informative comments**

49931. The TPM MAY support redirection

49942. If supported, the TPM MUST only use redirection on keys that have the redirect attribute  
4995 set

49963. A key that is tagged as a “redirect” key MUST be a leaf key in the TPM Protected Storage  
4997 blob hierarchy. A key that is tagged as a “redirect” key CAN NEVER be a parent key.

49984. Output data that is the result of a cryptographic operation using the private portion of a  
4999 “redirect” key:

- 5000 a. MUST be passed to an alternate output channel
- 5001 b. MUST NOT be passed to the normal output channel
- 5002 c. MUST NOT be interpreted by the TPM

50035. When command input or output is redirected the TPM MUST respond to the command  
5004 as soon as the ordinal finishes processing

- 5005 a. The TPM MUST indicate to any subsequent commands that the TPM is busy and  
5006 unable to accept additional command until the redirection is complete
- 5007 b. The TPM MUST allow for the resetting of the redirection channel

50086. Redirection MUST be available for the following commands:

- 5009 a. TPM\_Unseal
- 5010 b. TPM\_UnBind
- 5011 c. TPM\_GetPubKey
- 5012 d. TPM\_Seal
- 5013 e. TPM\_Quote

## 5014 **36. Structure Versioning**

### 5015 **Start of informative comment**

5016 In version 1.1 some structures also contained a version indicator. The TPM set the indicator  
5017 to indicate the version of the TPM that was creating the structure. This was incorrect  
5018 behavior. The functionality of determining the version of a structure is radically different in  
5019 1.2.

5020 Most structures will contain a TPM\_STRUCTURE\_TAG. All future structures must contain  
5021 the tag, the only structures that do not contain the tag are 1.1 structures that are not  
5022 modified in 1.2. This restriction keeps backwards compatibility with 1.1.

5023 Any 1.2 structure must not contain a 1.1 tagged structure. For instance the TPM\_KEY  
5024 complex, if set at 1.2, must not contain a PCR\_INFO structure. The TPM\_KEY 1.2 structure  
5025 must contain a PCR\_INFO\_LONG structure. The converse is also true 1.1 structures must  
5026 not contain any 1.2 structures.

5027 The TPM must not allow the creation of any mixed structures. This implies that a command  
5028 that deals with keys, for instance, must ensure that a complete 1.1 or 1.2 structure is  
5029 properly built and validated on the creation and use of the key.

5030 The tag structure is set as a UINT16. This allows for a reasonable number of structures  
5031 without wasting space in the buffers.

5032 To obtain the current TPM version the caller must use the TPM\_GetCapability command.

5033 The tag is not a complete validation of the validity of a structure. The tag provides a  
5034 reference for the structure and the TPM or caller is responsible for determining the validity  
5035 of any remaining fields. For instance, in the TPM\_KEY structure, the tag would indicate  
5036 TPM\_KEY but the TPM would still use tpmProof and the various digests to ensure the  
5037 structure integrity.

### 5038 7. Compatibility and notification

5039 In 1.1 TPM\_CAP\_VERSION (index 19) returned a version structure with 1.1.x.x. The x.x was  
5040 for manufacturer information and the x.x also was set version structures. In 1.2  
5041 TPM\_CAP\_VERSION will return 1.1.0.0. Any 1.2 structure that uses the version information  
5042 will set the x.x to 0.0 in the structure. TPM\_CAP\_MANUFACTURER\_VER (index 21) will  
5043 return 1.2.x.x. The 1.2 structures do not contain the version structure. The rationale  
5044 behind this is that the structure tag will indicate the version of the structure. So changing a  
5045 correct structure will result in a new tag and there is no need for a separate version  
5046 structure.

5047 For further compatibility, the quote function always returns 1.1.0.0 in the version  
5048 information regardless of the size of the incoming structure. All other functions may regard  
5049 a 2 byte sizeofselect structure as indicative of a 1.1 structure. The TPM handles all of the  
5050 structures according to the input, the only exception being TPM\_CertifyKey where the TPM  
5051 does not need to keep the input version of the structure.

### 5052 **End of informative comment**

5053 1. The TPM MUST support 1.1 and 1.2 defined structures

5054 2. The TPM MUST ensure that 1.1 and 1.2 structures are not mixed in the same overall  
5055 structure

- 5056 a. For instance in the TPM\_KEY structure if the structure is 1.1 then PCR\_INFO MUST  
5057 be set and if 1.2 the PCR\_INFO\_LONG structure must be set
50583. On input the TPM MUST ignore the lower two bytes of the version structure
50594. On output the TPM MUST set the lower two bytes to 0 of the version structure

## 5060**37. Certified Migration Key Type**

### 5061**Start of informative comment**

5062In version 1.1 there were two key types, non-migration and migration keys. The TPM would  
5063only certify non-migrating keys. There is a need for a key that allows migration but allows  
5064for certification. This proposal is to create a key that allows for migration but still has  
5065properties that the TPM can certify.

5066These new keys are “certifiable migratable keys” or CMK. This designation is to separate the  
5067keys from either the normal migration or non-migration types of keys. The TPM Owner is  
5068not required to use these keys.

5069Two entities may participate in the CMK process. The first is the Migration-Selection  
5070Authority and the second is the Migration Authority (MA).

### 5071**Migration Selection Authority (MSA)**

5072The MSA controls the migration of the key but does not handle the migrated itself.

### 5073**Migration Authority (MA)**

5074A Migration Authority actually handles the migrated key.

### 5075**Use of MSA and MA**

5076Migration of a CMK occurs using TPM\_CMK\_CreateBlob (TPM\_CreateMigrationBlob cannot  
5077be used). The TPM Owner authorizes the migration destination (as usual), and the key  
5078owner authorizes the migration transformation (as usual). An MSA authorizes the migration  
5079destination as well. If the MSA is the migration destination, no MSA authorization is  
5080required.

### 5081**End of informative comment**

## 5082**37.1 Certified Migration Requirements**

### 5083**Start of informative comment**

5084The following list details the design requirements for the controlled migration keys

### 5085**Key Protections**

5086The key must be protected by hardware and an entity trusted by the key user.

### 5087**Key Certification**

5088The TPM must provide a mechanism to provide certification of the key protections (both  
5089hardware and trusted entity)

### 5090**Owner Control**

5091The TPM Owner must control the selection of the trusted entity

### 5092**Control Delegation**

5093The TPM Owner may delegate the ability to create the keys but the decision must be explicit

### 5094**Linkage**

5095The architecture must not require linking the trusted entity and the key user



**5096Key Type**

5097The key may be any type of migratable key (storage or signing)

**5098Interaction**

5099There must be no required interaction between the trusted entity and the TPM during the  
5100key creation process

**5101End of informative comment****510237.2 Key Creation****5103Start of informative comment**

5104The command TPM\_CMK\_CreateKey creates a CMK where control of the migration is by a  
5105MSA or MA. The process uses the MSA public key (actually a digest of the MA public key) as  
5106input to TPM\_CMK\_CreateKey. The key creation process establishes a migrationAuth that is  
5107SHA-1(tpmProof || SHA-1(MA pubkey) || SHA-1(source pubkey)).

5108The use of tpmProof is essential to prove that CMK creation occurs on a TPM. The use of  
5109“source pubkey” explicitly links a migration AuthData value to a particular public key, to  
5110simplify verification that a specific key is being migrated.

**5111End of informative comment****511237.3 Migrate CMK to a MA****5113Start of informative comment**

5114Migration of a CMK to a destination other than the MSA:

**5115TPM\_MIGRATIONKEYAUTH Creation**

5116The TPM Owner authorizes the creation of a TPM\_MIGRATIONKEYAUTH structure using  
5117TPM\_AuthorizeMigrationKey command. The structure contains the destination  
5118migrationKey, the migrationScheme (which must be set to TPM\_MS\_RESTRICT\_MIGRATE  
5119or TPM\_MS\_RESTRICT\_APPROVE) and a digest of tpmProof.

**5120MA Approval**

5121The MA signs a TPM\_CMK\_AUTH structure, which contains the digest of the MA public key,  
5122the digest of the destination (or parent) public key and a digest of the public portion of the  
5123key to be migrated

**5124TPM Owner Authorization**

5125The TPM Owner authorizes the MA approval using TPM\_CMK\_CreateTicket and produces a  
5126signature ticket

**5127Key Owner Authorization**

5128The CMK owner passes the TPM Owner MA authorization, the MSA Approval and the  
5129signature ticket to the TPM\_CMK\_CreateBlob using the key owners authorization.

5130Thus the TPM owner, the key’s owner, and the MSA, all cooperate to migrate a key  
5131produced by TPM\_CMK\_CreateBlob.

**5132End of informative comment**

## 5133**37.4 Migrate CMK to a MSA**

### 5134**Start of informative comment**

5135Migrate CMK directly to a MSA

### 5136**TPM\_MIGRATIONKEYAUTH Creation**

5137The TPM Owner authorizes the creation of a TPM\_MIGRATIONKEYAUTH structure using  
5138TPM\_AuthorizeMigrationKey command. The structure contains the destination  
5139migrationKey (which must be the MSA public key), the migrationScheme (which must be set  
5140to TPM\_MS\_RESTRICT\_MIGRATE) and a digest of tpmProof.

### 5141**Key Owner Authorization**

5142The CMK owner passes the TPM\_MIGRATIONKEYAUTH to the TPM in a  
5143TPM\_CMK\_CreateBlob using the CMK owner authorization.

### 5144**Double Wrap**

5145If specified, through the MS\_MIGRATE scheme, the TPM double wraps the CMK information  
5146such that the only way a recipient can unwrap the key is with the cooperation of the CMK  
5147owner.

### 5148**Proof of Control**

5149To prove to the MA and to a third party that migration of a key is under MSA control, a  
5150caller passes the MA's public key (actually its digest) to TPM\_CertifyKey, to create a  
5151TPM\_CERTIFY\_INFO structure. This now contains a digest of the MA's public key.

5152A CMK be produced without cooperation from the MA: the caller merely provides the MSA's  
5153public key. When the restricted key is to be migrated, the public key of the intended  
5154destination, plus the CERTIFY\_INFO structure are sent to the MSA. The MSA extracts the  
5155migrationAuthority digest from the CERTIFY\_INFO structure, verifies that  
5156migrationAuthority corresponds to the MSA's public key, creates and signs a  
5157TPM\_RESTRICTEDKEYAUTH structure, and sends that signature back to the caller. Thus  
5158the MSA never needs to touch the actual migrated data.

### 5159**End of informative comment**

## 5160**38. Revoke Trust**

### 5161**Start of informative comment**

5162There are circumstances where clearing all keys and values within the TPM is either  
5163desirable or necessary. These circumstances may involve both security and privacy  
5164concerns.

5165Platform trust is demonstrated using the EK Credential, Platform Credential and the  
5166Conformance Credentials. There is a direct and cryptograph relationship between the EK  
5167and the EK Credential and the Platform Credential. The EK and Platform credentials can  
5168only demonstrate platform trust when they can be validated by the Endorsement Key.

5169This command is called revoke trust because, by deleting the EK, the EK Credential and the  
5170Platform Credential are dissociated from the platform, therefore invalidating them and  
5171resulting in the revocation of the trust in the platform. From a trust perspective, the  
5172platform associated with these specific credentials no longer exists. However, any  
5173transaction that occurred prior to invoking this command will remain valid and trusted to  
5174the same extent they would be valid and trusted if the platform were physically destroyed.

5175This is a non-reversible function. Also, along with the EK, the Owner is also deleted,  
5176removing all non-migratable keys and owner-specified state.

5177It is possible to establish new trust in the platform by creating a new EK using the  
5178TPM\_CreateRevokableEK command. Establishing trust in the platform, however, is more  
5179than just creating the EK. The EK Credential and the Platform Credential must also be  
5180created and associated with the new EK as described above. (The conformance credentials  
5181may be obtained from the TPM and Platform manufacturer.) These credentials must be  
5182created by an entity that is trusted by those entities interested in the trust of the platform.  
5183This may not be a trivial task. For example, an entity willing to create these credentials my  
5184want to examine the platform and require physical access during the new EK generation  
5185process.

5186Besides calling one of the two EK creation functions to create the EK, the EK may be  
5187"squirted" into the TPM by an external source. If this method is used, tight controls must be  
5188placed on the process used to perform this function to prevent exposure or intentional  
5189duplication of the EK. Since the revocation and re-creation of the EK are functions intended  
5190to be performed after the TPM leaves the trusted manufacturing process, squirting of the EK  
5191must be disallowed after the manufacturing process if the revoke trust command is  
5192executed.

### 5193**End of informative comment**

51941. If TPM\_CreateRevokableEK and TPM\_RevokeTrust are implemented, one can do an  
5195unrestricted number of TPM\_CreateRevokableEK / TPM\_RevokeTrust pairs until  
5196TPM\_CreateEndorsementKeyPair is called. After TPM\_CreateEndorsementKeyPair is  
5197called, the EK becomes irrevocable.

51982. After an EK is created the TPM MUST NOT allow a new EK to be "squirted" for the  
5199lifetime of the TPM.

52003. The EK Credential MUST provide an indication within the EK Credential as to how the  
5201EK was created. The valid permutations are:

5202a. Squirted, non-revocable

5203 b. Squirted, revocable

5204 c. Internally generated, non-revocable

5205 d. Internally generated, revocable

52064. If the method for creating the EK during manufacturing is squirting, the EK may be  
5207 either non-revocable or revocable. If it is revocable, the method must provide the  
5208 insertion or extraction of the EKreset value.

## 5209**39. Mandatory and Optional Functional Blocks**

### 5210**Start of informative comment**

5211This section lists the main functional blocks of a TPM (in arbitrary order), states whether  
5212that block is mandatory or optional in the main TPM specification, and provides brief  
5213justification for that choice.

5214Important notes:

52151. The default classification of a TPM function block is “mandatory”, since reclassification  
5216from mandatory to optional enables the removal of a function from existing  
5217implementations, while reclassification from optional to mandatory may require the addition  
5218of functionality to existing implementations.

52192. Mandatory functions will be reclassified as optional functions if those functions are not  
5220required in some particular type of TCG trusted platform.

52213. If a functional block is mandatory in the main specification, the functionality must be  
5222present in all TCG trusted platforms.

52234. If a functional block is optional in the main specification, each individual platform-  
5224specific specification must declare the status of that functionality as either (1) “mandatory-  
5225specific” (the functionality must be present in all platforms of that type), or (2) “optional-  
5226specific” (the functionality is optional in that type of platform), or (3) “excluded-specific” (the  
5227functionality must not be present in that type of platform).

### 5228**End of informative comment**

5229Classification of TPM functional blocks

52301. Legacy (v1.1b) features

5231 a. Anything that was mandatory in v1.1b continues to be mandatory in v1.2. Anything  
5232 that was optional in v1.1b continues to be optional in v1.2.

5233 b. V1.2 must be backwards compatible with v1.1b. All TPM features in v1.1b were  
5234 discussed in depth when v1.1b was written, and anything that wasn't thought  
5235 strictly necessary was tagged as "optional".

52362. Number of PCRs

5237 a. The platform specific specification controls the number of PCR on a platform. The  
5238 TPM MUST implement the mandatory number of PCR specified for a particular  
5239 platform

5240 i. TPMs designed to work on multiple platforms MUST provide the appropriate  
5241 number of TPM for all intended platforms. I.e. if one platform requires 16 PCR  
5242 and the other platform 24 the TPM would have to supply 24 PCR.

5243 b. For TPMs providing backwards compatibility with 1.1 TPM on the PC platform, there  
5244 MUST be 16 static PCR.

52453. Sessions

5246 a. The TPM MUST support a minimum of 3 active sessions

5247 i. Active means currently loaded and addressable inside the TPM

- 5248 ii. Without 3 active sessions many TPM commands cannot function
- 5249 b. The TPM MUST support a minimum of 16 concurrent sessions
- 5250 i. The contextList of currently available session has a minimum size of 16
- 5251 ii. Providing for more concurrent sessions allows the resource manager additional
- 5252 flexibility and speed
- 5253 4. NVRAM
- 5254 a. There are 20 bytes mandatory of NVRAM in v1.2 as specified by the main
- 5255 specification. A platform specific specification can require a larger amount of NVRAM
- 5256 b. Cost is important. The mandatory amount of NVRAM must be as small as possible,
- 5257 because different platforms will require different amounts of NVRAM. 20 bytes are
- 5258 required for (DIR) backwards compatibility with v1.1b.
- 5259 5. Keys
- 5260 a. The new signing keys are mandatory in v1.2 because they plug a security hole.
- 5261 b. The TPM must support a minimum of 2 key slots.
- 5262 6. Direct Anonymous Attestation
- 5263 a. This is optional in v1.2
- 5264 b. Cost is important. The DAA function consumes more TPM resources than any other
- 5265 TPM function, but some platform specific specifications (some servers, for example)
- 5266 may have no need for the anonymity and pseudonymity provided by DAA.
- 5267 7. Transport sessions
- 5268 a. These are mandatory in v1.2.
- 5269 b. Transport sessions
- 5270 i. Enable protection of data submitted to a TPM and produced by a TPM
- 5271 ii. Enable proof of the TPM commands executed during an arbitrary session.
- 5272 8. Resettable Endorsement Key
- 5273 a. This is optional in v1.2
- 5274 b. Cost is important. Resettable EKs are valuable in some markets segments, but cause
- 5275 more complexity than non-resettable EKs, which are expected to be the dominant
- 5276 type of EK
- 5277 9. Monotonic Counter
- 5278 a. This is mandatory in v1.2
- 5279 b. A monotonic counter is essential to enable software to defeat certain types of attack,
- 5280 by enabling it to determine the version (revision) of dynamic data.
- 5281 10. Time Ticks
- 5282 a. This is mandatory in v1.2
- 5283 b. Time stamping is a function that is potentially beneficial to both a user and system
- 5284 software.

## 528511.Delegation (includes DSAP)

5286 a. This is mandatory in v1.2

5287 b. Delegation enables the well-established principle of least privilege to be applied to  
5288 Owner authorized commands.

## 528912.GPIO

5290 a. This is optional in v1.2

5291 b. Cost is important. Not all types of platform will require a secure intra-platform  
5292 method of key distribution

## 529313.Locality

5294 a. The use of locality is optional in v1.2

5295 b. The structures that define locality are mandatory

5296 c. Locality is an essential part of many (new) TPM commands, but the definition of  
5297 locality varies widely from platform to platform, and may not be required by some  
5298 types of platforms.

5299 d. It is mandatory that a platform specific specification indicate the definitions of  
5300 locality on the platform. It is perfectly reasonable to only define one locality and  
5301 ignore all other uses of locality on a platform

## 530214.TPM-audit

5303 a. This is optional in v1.2

5304 b. Proper TPM-audit requires support to reliably store logs and control access to the  
5305 TPM, and any mechanism (an OS, for example) that could provide such support is  
5306 potentially capable of providing an audit log without using TPM-audit. Nevertheless,  
5307 TPM-audit might be useful to verify operation of any and all software, including an  
5308 OS. TPM-audit is believed to be of no practical use in a client, but might be valuable  
5309 in a server, for example.

## 531015.Certified Migration

5311 a. This is optional in v1.2

5312 b. Cost is important. Certified Migration enables a business model that may be  
5313 nonsense for some platforms.

## 5314**40. 1.1a and 1.2 Differences**

### 5315**Start of informative comment**

5316All 1.2 TPM commands are completely compliant with 1.1b commands with the following  
5317known exceptions.

53181. TSC\_PhysicalPresence does not support configuration and usage in a single step.

53192. TPM\_GetPubKey is unable to read the SRK unless TPM\_PERMANENT\_FLAGS ->  
5320readSRKPub is TRUE

53213. TPM\_SetTempDeactivated now requires either physical presence or TPM Operator  
5322authorization to execute

53234. TPM\_OwnerClear does not modify TPM\_PERMANENT\_DATA -> authDIR[0].

### 5324**End of informative comment**