Tamper free deployment and execution of software using TPM

Michael Emanuel
February 12, 2012

Abstract

This report will provide a method for achieving copy protection of the protected software. This protection scheme will secure the software in the stage of installation, as well as during execution. The method will also offer protection of Intellectual Property, as the software is kept hidden from unwanted audience. The protective method is based on the security building blocks offered by the Trusted Platform Module (TPM) cryptographic chip.

Table of Contents

1 Motivation....4
2 Problem statement....5
2.1 Only Software protection, not hardware based attacks....5
2.2 What does tamper free cover....5
2.3 What is TPM, and is it a good conveyer of trust....6
3 Method....8
4 Related work....10
4.1 One-Way Function (OWF)....10
4.2 Collision Resistant (CR)....10
4.3 Merkle-Damgård Construction (MDC)....11
4.4 Compression Function (C)....12
4.5 Trapdoor One-Way Permutation (TOWP)....13
4.6 Optimal Asymmetric Encryption Padding (OAEP)....13
4.7 Cipher-text Indistinguishable (IND)....14
4.8 Indistinguishable under chosen plain-text attack (IND-CPA)....14
4.9 Trusted Computing Base (TCB)....15
4.10 Root of Trust....15
4.11 Key management and creation on TPM....16
4.12 Locality on TPM....17
4.13 TPM States....19
5 Analyzes and Results....21
5.1 Tamper free installation....21
5.1.1 Detailed description of how to achieve Tamper free installation....22
5.1.1.1 Validate the TPM using Remote Attestation....22
5.1.1.2 Object-Specific Authorization Protocol (OSAP)....23
5.1.1.3 Transport Session....25
5.1.1.4 Create a SEALED package of the protected software....28
5.1.2 Attack methods on Tamper free installation....31
5.1.2.1 Man In The Middle (MITM) Attack....31
5.1.2.2 Rogue TPM Attack....32
5.1.2.3 Brute-Force Attack on TPM....32
5.1.2.4 Replay Attack....33
5.2 Tamper free execution....34
5.2.1 Detailed description of how to achieve Tamper free execution....35
5.2.1.1 SENTER on Intel Family CPU....35
5.2.1.2 TPM_Unseal on TPM....39
5.2.1.3 Exit Protected Execution....41
5.2.2 Attack methods on Tamper free execution....42
5.2.2.1 Read Protected software from SEALED package....42
5.2.2.2 UNSEAL on Other Platform....42

5.2.2.3 UNSEAL from Other Software 42
5.2.2.4 UNSEAL without Protected Execution....42
5.2.2.5 Memory Access....43
5.2.2.6 Attack SINIT....44
5.2.2.7 Attack TPM device....45
5.2.2.8 Attack by Reset 46
5.2.2.9 Attack by Change of Power Level....47
5.3 Future work....49
6 Conclusion....52
References....54
Glossary....55
Appendix....56
A1 - TPM Commands and Structures....56
A2 – Overview of tamper free method....63

1 Motivation

Most of the copy protection methods that are in use today seem more like ways to annoy the legitimate consumers than ways to hinder illegal copying.
The more a copy protection method will hinder any legitimate use of the software by the consumer, and the less effective the method is to fulfill its purpose of preventing any illegal copying, the more annoying this method seems to me as a consumer.
Paying an unnecessary high price for a piece of software, because the few legitimate consumers must pay the price that aught to be payed by the large group of free-riders that use illegal copies of the software for free. This is another way that the lack of efficient copy protection methods can be an annoyance to the consumer.

As a software developer the existence of efficient copy protection mechanisms is a way to ensure that you can earn your monthly salary. If illegal software copying makes it impossible to earn money on selling software, then a large part of the software industry that uses this type of business model will be unable to make a profit. Without a way to effectively inhibit illegal copying of software, then any business model that relies on purchasing software will be threatened. Many types of software products and many lines of business can only fit into a purchase oriented business model. Without an efficient way to protect against illegal software copying, the diversity of software types and business lines will decline.

From the viewpoint of an IT security professional, solving the problem of illegal copying of software pose some very interesting academical challenges.
There are IT security related challenges to analyze possible types of attack to find any possible ways an attacker can circumvent the protection mechanisms.
And there are the cryptographic countermeasures to block these attacks and keep the protection mechanisms intact.
The derived protection method can be a combination of well-known and proven defense mechanisms as well as new and unproved methods of defense.

From a company's viewpoint as well as for the software business as a hole, there are large economical gains in finding methods to enable repelling illegal software copying.
Large amounts of money are lost as a result of software piracy. An effective method of preventing illegal copying of software that could be deployed on the type of hardware that are broadly available to the public (that be personal computers, laptops, mobile phones, or other types of processor equipped hardware) would represent a very interesting opportunity for any company that are oriented towards a purchase based business model.

2 Problem statement

Tamper free deployment and execution of software using TPM

2.1 Only Software protection, not hardware based attacks

This report will only address protection against software based attacks.
That is attacks which will be performed by execution of code.
Methods of attack which involve directly measuring or modifying the hardware that the protected code is exposed to is not covered by this report.
Examples of types of hardware attacks not covered:

- Attach probe and watch the Front Side Bus (FSB).
- Freeze down and move system memory to an unsecured computer.

2.2 What does tamper free cover

The term tamper free, will in this context be a method to avoid the protected code to be moved, duplicated or modified by an attacker, without the action is detected. The protected code must also be kept secret, in such a way that an attacker must not be able to view the code in order to duplicate the code or any part of it.
The method must have the ability to avoid the following actions from an attacker:

- Modify the protected code without the modification is detectable.
- View the protected code to enable copying it.
- Execute the protected code on another platform.

The tamper free protection must be effective in all stages or environments:

- Transferred on the internet or other media as an installation package
- Before installation, stored on a storage device
- During execution of installation
- Located on a storage device after installation
- Located in memory during execution

In practice this method will provide a copy protection of the protected software and a protection of Intellectual Property as the code is kept hidden from unwanted audience.

2.3 What is TPM, and is it a good conveyer of trust

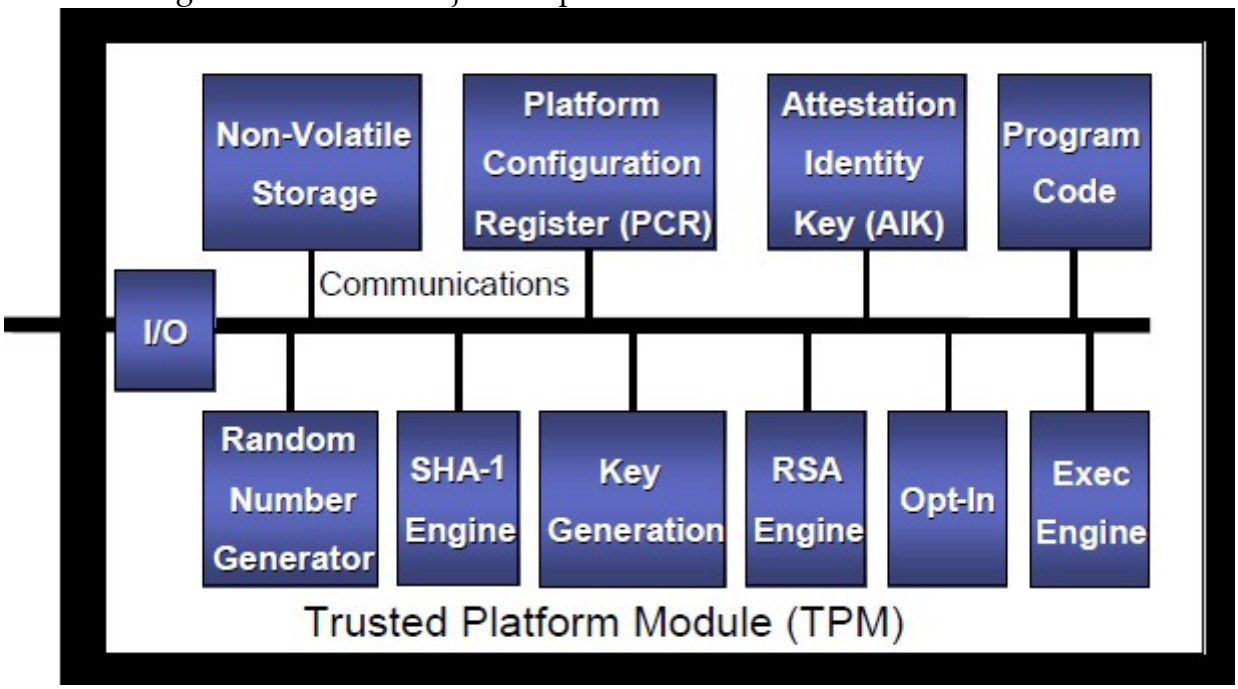
In this section a short description of the Trusted Platform Module (TPM) is given.

- **Physical presence**
  TPM is a microchip with a build-in SHA Engine, Cryptographic Key Generator and Non-volatile memory. The structure of TPM is described in detail in [1].
  It is available to a wide range of different platforms like: laptops, computers servers, mobile phones, storage devices, network devices and embedded systems.
- **Usage**
  The main purpose of TPM is to ensure that the Platform can be trusted. The principles behind TPM is described in [2]. The TPM uses various Trust Building Blocks to reach this goal.
  These building blocks are:
  - 2048-bit RSA key pair called the Endorsement Key (EK) is a non-migratable asymmetric key. The private portion of the key exists only in a TPM-shielded location. EK can be used as an attestation identity key (AIK). A Certificate with a Trusted Third Party's Signature to perform Attestation of the TPM Validity.
  - The TPM implements protected capabilities and shielded-locations used to protect and report integrity measurements called Platform Configuration Registers (PCR). A PCR is a 160-bit storage location for discrete integrity measurements. The SHA-1 Engine is used to perform Hash of objects to measure, storing the integrity measurements into PCR registers. The measured object could be a piece of software like the BIOS, a driver or an application. The measured object could also be some platform configuration or a piece of data. All PCR registers are shielded locations and are inside of the TPM.
  - Sealed Storage
  Sealed messages are bound to a set of platform metrics specified by the message sender. Platform metrics specify the platform configuration state that must exist before decryption will be allowed. Sealing associates the encrypted message (actually the symmetric key used to encrypt the message) with a set of PCR register values and a non-migrateable asymmetric key.
- **Widespread**
  The TPM specifications is developed and published by Trusted Computing Group (TCG) : http://www.trustedcomputinggroup.org.
  According to TCG about 300,000,000 PCs have shipped with a TPM, and virtually every business laptop comes with an embedded TPM chip.
  TPM is supported by several vendors: Acer, Asus, Dell, LG, Fujitsu, HP, Lenovo, Samsung, Sony and Toshiba provide TPM integration on their devices.
  It is possible to find computer models sold on the internet by the above manufacturers that provide TPM integration. It is primarily on the models oriented towards the business segment that TPM is present.

1 TPM Main-Part 2 TPM Structures_v1.2
2 TPM Main-Part 1 Design Principles_v1.2

- **Partners**
  Members of TCG are:
  - Microsoft
  - Intel Corporation
  - AMD
  - IBM
  - Lenovo
  - HP
  - Fujitsu
  - Infineon
- **A Platform of Trust?**
  TPM seems to have the cryptographic and architectural capabilities to ensure trust. TPM-shielded location of EK and PCR protects against tampering and the RSA Engine ensures a cryptographic strong encryption.
  TCG has support from some of the most important partners in the industry. This ensures that the initiative will not vanish over night.
  The widespread use of the TPM in the PC segment, and the broad support of different platform types from mobile phones to corporate servers, make an implementation using the TPM practically possible.
  The conclusion is that the TPM is a good provider of trust.

This block diagram shows the major components of the TPM:



3 Method

The goal of this report
The goal of this report is to confirm the following hypothesis.

Hypothesis: Using the TPM it will be possible to design a good mechanism that protects against tampering with the installation and execution of the protected software.

The tamper protection refer to providing a copy protection of the protected software and a protection of Intellectual Property by keeping the code hidden from unwanted audience, as described in the problem statement section.

The aim is not to make it mathematically provable impossible to bypass the protection mechanism, obtaining such a proof would be very ambitious when it comes to software. If only it is computationally hard for an attacker to break the protection mechanism, then the goal of designing a good protection mechanism is considered successful.

The product of and the verification method used by this report
The product of this report will not be any code examples or a produced prototype. The report will derive a method to solve the problem stated in the problem statement and perform a theoretical description of the method.
Verification of the hypothesis will be achieved by analyzing the possible attack methods against the proposed protection mechanism and argue or mathematically prove that the proposed method will protect against this attack.

Platform dependency of this report
Most of the TPM specifications are of a platform independent nature. For those parts of the specification that have platform dependencies, the scope of this report is chosen to be the Computer platform (covering devices like laptop, PC or servers). The reason for this choice is that the computer platform seems to be the target where a tamper free protection mechanism would be of most use.

CPU Family dependency of this report
The dominating CPU families on the computer platform are manufactured by Intel and AMD. Both of these two CPU families support a Protected Execution Scheme: SENTER[4] on Intel family CPU's and SKINIT[5] on AMD family CPU's. Certain security properties of the proposed protection scheme rely on the security properties of the protected execution scheme. As the time frame for producing this report have been limited, a choice had to be made between a thorough and detailed analysis of the security properties of the Protected Execution scheme for one of the CPU families, or a more wide but shallow analysis

4 The Intel Safer Computing Initiative
5 AMD64 Architecture Programmers Manual

covering both CPU families. The informations published by AMD about the SKINIT operation are not very detailed. The security properties of the measurement and execution of the Measured Virtual Machine Monitor (referred to as the secure loader) is described as, quote: “starts execution of the secure loader (SL) in a way that cannot be tampered with.” A search for further documentation revealed numerous papers referring to the SKINIT operation, but reading these documents reveals no further details about the exact methods used by the SKINIT operation to achieve the tamper free properties mentioned in the published material from AMD. A detailed analysis of the security properties of the AMD SKINIT operation would require some amount of reverse engineering to reveal the actual security measures taken by AMD. The bottom line is that a detailed analysis based on the more superficial publications from AMD would require investing a great amount of time due to the necessary of reverse engineering; in contrast to the more detailed publications from Intel that enable a theoretical analysis of the security measures based on the published documentation. The choice has been made to perform a detailed analysis of the protected execution scheme used by Intel.

4 Related work

In this section various cryptographic definitions are described.
The rest of the report will reference to and build upon these definitions. Sections in the remainder of this report will be referring to concepts described in this section. It is assumed that the reader will be familiar with the concept in a degree that corresponds to the level of detail with which the concept has been described in this section.

4.1 One-Way Function (OWF)

If a function $H : \{0,1\}^{512} \rightarrow \{0,1\}^{160}$ is SHA-1, we think it is computationally hard to invert H on uniformly random inputs.

> Definition 1: For any integer $t \in N$, probability $\varepsilon \in [0,1]$ and function $f : \{0,1\}^{L} \rightarrow \{0,1\}^{L'}$ we say that f is a (t,ε)-OWF (one-way-function) if it holds for all algorithms A running in time t that $Pr[f(x') = y] \leq \varepsilon$ when $y = H(x)$ for a uniformly random $x \in \{0,1\}^{L}$ and $x' = A(y)$

A one-way-function is a function which is easy to compute in the forward direction, but which is hard to invert, that is compute in the backwards direction. For a function to be called a OWF, it should be a (t,ε)-OWF for a very small ε and a very huge t (eg. $\varepsilon = 2^{-80}$ and $t = 2^{160}$)
It is a desired property for a good cryptographic hash function to be a good OWF.

4.2 Collision Resistant (CR)

A function $H : \{0,1\}^{\leq L} \rightarrow \{0,1\}^{L'}$ is called collision resistant if it is hard to find $x \in \{0,1\}^{L}$ and $x' \in \{0,1\}^{L}$ such that $x \neq x'$ and $H(x) = H(x')$ – the value (x,x') is a collision. If $L \leq L'$ then a collision exists, we only require that they are computationally hard to find.

> Definition 2: Let $H\{0,1\}^{K} : \{0,1\}^{\leq L} \rightarrow \{0,1\}^{L'}$ and let A be an algorithm.
> Sample uniformly $K \in \{0,1\}^{K}$ and compute $(x,x') \leftarrow A(K)$. We say that A finds a collision if $x,x' \in \{0,1\}^{\leq L}$ and $x \neq x'$ and $H_K(x) = H_K(x')$. Then H is (t,ε)-CR (Collision Resistant) if it holds for all A running in time most t has the probability to find a collision $\leq \varepsilon$.

4.3 Merkle-Damgård Construction (MD_C)

The Secure Hash Algorithm (SHA-1) uses a structure known as Merkle-Damgård construction (MD_C). The Merkle-Damgård construction uses a so-called compression function which itself is a collision resistant hash function, but with a relatively short input length, and extends it to a collision resistant hash function with a much longer collision length.

In the case of SHA-1 the compression function C takes 672 bits as input and delivers 160 bits of output, It is convenient to think of C taking two inputs of 160 and 512 bits and deliver one output of length 160, i.e. $C : \{0,1\}^{160} \times \{0,1\}^{512} \rightarrow \{0,1\}^{160}$.

The compression function starts with a initialization vector $h0 \in \{0,1\}^{160}$. It then splits the input m into B blocks of length 512 bits (m1..mB). Then it computes h1 = C(h0, m1), h2 = C(h1, m2), … , hB = C(hB-1, mB). The resulting output is h = hB.

To handle messages of arbitrary length securely, some carefully chosen padding is used. First, to make the length of m a multiple of 512, we pad it with 0's. If $|m|$ is the length of m and we append $-|m| \bmod 512$ zero-bits, then the new length will be $|m| + (-|m| \bmod 512)$, we see that

$|m| + (-|m| \bmod 512) \bmod 512 = |m| + (-|m|) \bmod 512 = 0$, so 512 divides the length.

Then wee add new 512-bit block which contain the binary representation of the message length $|m|$. We denote this block $<|m|>_{512}$. The resulting message M consists of some B blocks of 512 bits in length.

The security of the Merkle-Damgård construction comes from the fact that one cannot find a collision (m,m') for $H = MD_C$ without finding a collision for the compression function C, so the construction is at least as secure as C.

In the case where m and m' have different length, the last block of m and m' (mB and m'B) will be different.

If we have a collision (m,m') then H(m) = H(m').

Since	H(m) = hB and H(m') = h'B
it follows that	C(hB-1, mB) = hB = h'B = C(h'B-1, m'B).
From	$mB \neq m'B$
it follows that	$(hB-1, mB) \neq (h'B-1, m'B)$.

So (hB-1, mB), (h'B-1, m'B) is a collision for C.

If m and m' have the same length, and assume the messages are different but H(m) = H(m'). Then (m1..mB) and (m'1..m'B) will also be of the same length, and be different, and hB = h'B.

Note that	hi = C(hi-1, Mi) and h'i = C(h'i-1, M'i) for all i = 1..B.

We claim that for some i the pair ((hi-1,Mi), (h'i-1,M'i)) is a collision for C.

The argument goes:

Since	$m \neq m'$
we have that	Mi ≠ M'i for at least one i,
so	$(hi-1,Mi) \neq (h'i-1,M'i)$ for at least one i.
If	C(hi-1,Mi) = C(h'i-1,M'i) then we found a collision.

If C(hi-1,Mi) ≠ C(h'i-1,M'i)
then hi ≠ h'i as hi = C(hi-1,Mi) and h'i = C(h'i-1,M'i).
But if hi ≠ h'i then (hi,Mi+1) ≠ (h'i,M'i+1).
If C(hi,Mi+1) = C(h'i,M'i+1) then we found a collision.
If not then hi+1 ≠ h'i+1
as hi+1 = C(hi,Mi+1) and h'i+1 = C(h'i,M'i+1).
Since at some point for j = i + 1...B it must happen that hj = h'j (as hB = h'B) we must at some point find a collision for H if we keep repeating this argument.

Note that for the above argument to work all we need was that if m and m' were of different lengths then the last block in their padded version becomes different, and that two different messages of the same length becomes padded to different messages of the same length. SHA-1 uses this fact to do a slightly optimized padding which often avoids adding an extra block: the length |x| is written into the padded zero-bits if there is room. The same argument as above applies: finding a collision for SHA-1 involves finding a collision for the compression function.

4.4 Compression Function (C)

The SHA-1 Compression Function maps two inputs $s \in \{0,1\}^{160}$ and $m \in \{0,1\}^{512}$ to a message digest $h \in \{0,1\}^{160}$.

The following pseudo code describes the SHA-1 Compression Function h = C(s, m):

```
(A,B,C,D,E) ← (s1,s2,s3,s4,s5) where s_i ∈ {0,1}^32 ← s.
(m1,m2,..,m16) where m_i ∈ {0,1}^32 ← m.
for i = 17 to 80 do: m_i ← (m_i-3 XOR m_i-8 XOR m_i-14 XOR m_i-16) << 1;
for i = 1 to 80 do:
{
      if 1 ≤ i ≤ 20 then
      {
            F ← (B AND C) OR (!B AND D)
            K ← 0x5A827999
      }
      if 21 ≤ i ≤ 40 then
      {
            F ← B XOR C XOR D
            K ← 0x6ED9EBA1
      }
      if 41 ≤ i ≤ 60 then
      {
            F ← (B AND C) OR (B AND D) OR (C AND D)
            K ← 0x8F1BBCDC
      }
      if 61 ≤ i ≤ 80 then
      {
            F ← B XOR C XOR D
            K ← 0xCA62C1D6
      }
      temp ← (A << 5) + F + E + K + m_i mod 2^32;
      E ← D;
      D ← C;
```

```
      C ← B << 30;
      A ← temp
}
s1 ← s1 + A mod 2^32;
s2 ← s2 + B mod 2^32;
s3 ← s3 + C mod 2^32;
s4 ← s4 + D mod 2^32;
s5 ← s5 + E mod 2^32;
h ← (s1,s2,s3,s4,s5)
return h
```

A method have been found to find collisions for SHA-1 in time about 2^{63} according to[6].

4.5 Trapdoor One-Way Permutation (TOWP)

A trapdoor permutation consists of three efficient algorithms (G, f, f^{-1}).
G is the key generator for sampling a random key pair (pubkey, s) ← G where pubkey is the public key and s is the trapdoor.
The algorithm f is deterministic. It takes a public key pubkey as input along with an element x Є $\{0,1\}^L$. It outputs $y = f_{pubkey}(x)$ where y Є $\{0,1\}^L$.
The algorithm f^{-1} is deterministic. It takes a trapdoor s as input along with an element y Є $\{0,1\}^L$. It outputs $x = f^{-1}{}_s(y)$ where x Є $\{0,1\}^L$.
To be a TOWP it must be hard to compute $y = f^{-1}{}_s(x)$ on a uniformly random x, when pubkey is a random public key and one does not know s. So f_{pubkey} should be a OWF to anyone who does not know the trapdoor s corresponding to pubkey.

Definition 5: For a (t,ε)-TOWP it must hold for all algorithms A running in time t that $Pr[x' = x] \leq \varepsilon$ when (pubkey, t) ← G is sampled at random, x Є $\{0,1\}^L$ is sampled uniformly at random, $y = f_{pubkey}(x)$ and $x' = A(pubkey, y)$ and $x = f^{-1}{}_s(y)$

A Trapdoor permutation can be thought of as a keyed OWF which is a bijection between two sets of the same size, which is easy to invert if you know the trapdoor.

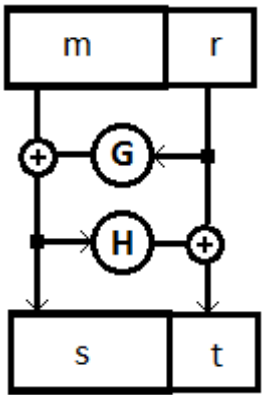
4.6 Optimal Asymmetric Encryption Padding (OAEP)

RSA use a secure padding method called OAEP.

m is the message to encrypt.
r is a random padding
G : Expands $\{0,1\}^{L1} \rightarrow \{0,1\}^{L2}$
H : Reduces $\{0,1\}^{L2} \rightarrow \{0,1\}^{L1}$

OAEP(m,r) = s||t = (m XOR G(r) || (r XOR H(m XOR G(r))))

6 Martin Cochran, 2008, Notes on the Wang et al. 2^63 SHA-1 Differential Path



4.7 Cipher-text Indistinguishable (IND)

We want the cryptosystem to hide all individual bits of the message well, and that it does so for all messages, without assuming anything about the distribution of the messages. If an adversary is unable to distinguish which is the encrypted cipher-text, then the algorithm is IND.

Definition 3: Let A_0 and A_1 be two interactive algorithms. We define the advantage of D in distinguishing A_0 and A_1:
$ADV_D(A_0, A_1) = |Pr[D(A_0) = 1] - Pr[D(A_1) = 1]|$
Let $t \in N$ be a running time and let $\varepsilon \in [0,1]$ be a probability. We say that A_0 and A_1 are are (t,ε)-IND if it holds for all experimenters D running in time t that $ADV_D(A_0, A_1) \leq \varepsilon$.

4.8 Indistinguishable under chosen plain-text attack (IND-CPA)

If the cryptosystem is Indistinguishable when we let the attacker pick the plain-text, then the algorithm is IND-CPA.

Definition 4: Let G be an algorithm which generates a public key pubkey and let E be a function which takes three inputs: pubkey, $m \in \{0,1\}^K$ and $r \in \{0,1\}^L$ and outputs $c = E_{pubkey}(m; r)$. Consider the following algorithm G_b parametrized by a bit $b \in \{0,1\}$: It first generates pubkey $\leftarrow G(s)$ for some uniformly random s, and it outputs pubkey. Then it waits for an input (m_0, m_1) where $m_0, m_1 \in \{0,1\}^K$. Then it computes $c = E_{pubkey}(m_b; r)$ for a uniformly random $r \in \{0,1\}^L$ and outputs c. We say that (G, E) if (t,ε)-IND-CPA if G_0 and G_1 are (t,ε)-IND

For a function to be called IND-CPA, it should be a (t,ε)-IND-CPA for a very small ε and a very huge t (eg. $\varepsilon = 2^{-80}$ and $t = 2^{160}$)
As demonstrated in [7]it can be shown that OAEP encryption with a TOWP like RSA is IND-CPA secure.

7 Kiltz et.al, 2010, Instantiability of RSA-OAEP under Chosen-Plaintext Attack

4.9 Trusted Computing Base (TCB)

Trusted Computing Base (TCB) is the part of the system that is secure even during an attack. An assumption is that if TCB is kept secure, then the rest of the system will be secure. By contrast will a bug or a vulnerability in TCB jeopardize the security properties of the entire system. A smaller TCB increase the probability and reduce the costs in keeping the TCB secure. In practice it is thus an aim to keep the TCB as small as possible.

4.10 Root of Trust

The Core Root of Trust for Measurement (CRTM) is the instructions executed by the platform when it acts as the Root of Trust for Measurement (RTM). The RTM is also the root of the chain of transitive trust. The CRTM is assumed secure and trustworthy even if the system should be attacked, as such the CRTM can be seen as part of the TCB.

Trusted Building Blocks (TBB) are the parts of the Roots of Trust that do not have shielded locations or protected capabilities. Normally these include just the instructions for the RTM and TPM initialization functions (reset, etc.). One example of a TBB is the combination of the CRTM, connection of the CRTM storage to a motherboard, the connection of the TPM to a motherboard, and mechanisms for determining Physical Presence. The TBB is trusted, meaning it is expected to behave in a way that doesn't compromise the goals of trusted platforms.

Transitive trust also known as "Inductive Trust", is a process where the Root of Trust gives a trustworthy description of a second group of functions. Based on this description, an interested entity can determine the trust it is to place in this second group of functions. If the interested entity determines that the trust level of the second group of functions is acceptable, the trust boundary is extended from the Root of Trust to include the second group of functions. In this case, the process can be iterated. The second group of functions can give a trustworthy description of the third group of functions, etc. Transitive trust is used to provide a trustworthy description of platform characteristics, and also to prove that non-migrateable keys are non-migrateable. Transitive trust is applied to a system booting from a static root of trust and the trust boundary is extended to include code that didn't natively reside within the roots of trust. In each extension of the trust boundary, the target code is first measured before execution control is transferred.
The Root of Trust is a mechanism to extend the trust placed in CRTM to other components in the system that wouldn't otherwise be seen as part of TCB.

Transitive trust applied to system boot from a static root of trust:

1. BIOS ROM (CRTM code)
2. OS Loader code
3. OS code
4. Application code

A more detailed description of how the Root of Trust is implemented using TPM on the PC platform can be found in [9]

4.11 Key management and creation on TPM

Asymmetric Keys can be used by the TPM for two purposes :

- sign/verify or
- encrypt/decrypt.

The different type of Keys can be used to perform different functions.
Sign : TPM_KEY_SIGNING and TPM_KEY_IDENTITY.
Encrypt : TPM_KEY_STORAGE, TPM_KEY_BIND, TPM_KEY_MIGRATE, and TPM_KEY_AUTHCHANGE.
The migrate-ability of Keys can be : Non-migrateable Key, Migrateable Key or Certifiable Migrateable Key.
A non-migratable key is permanently associated with a specific TPM instance and cannot be exchanged between TPM devices.

The Key Hierarchy can be displayed like this:
Endorsement Key (EK)
Storage Root Key (SRK)
→ Attestation Key (AIK)
→ Non-migrateable Storage Key
→ Non-migrateable Signing Key
→ Non-migrateable Encryption Key
→ Non-migrateable Storage Key
→ Migrateable Signing Key
→ Migrateable Encryption Key
→ Migrateable Storage Key
→ Migrateable Storage Key
→ Migrateable Signing Key
→ Migrateable Encryption Key
→ Migrateable Storage Key

A short description of the structure of the key hierarchy follows here:
The private part of EK and the SRK will never leave the TPM. The private part of these keys exists only in a TPM-shielded location.
As the amount of Key Slots within the TPM are limited, then a mechanism must exist to securely export a key from the TPM while still preserving the integrity of the key.
Likewise a mechanisms must exist to load a key from external storage.
These operations are performed by the TPM_CreateWrapKey and TPM_LoadKey commands, the management of key slots is performed by the TGC Software Stack (TSS), a software section that is external to the TPM.

9 TCG_PCClientTPMSpecification_v1-21

When exporting a key from the TPM, the private part of the key is encrypted with its public parent key. This ensures that access to the clear-text of the security sensitive private part of an exported key will require access to the private part of the parent key. Without the parent key it will not be computationally feasible to decrypt the exported key blob. The child key is said to be wrapped by its parent key in the key hierarchy. Moving up the key hierarchy, the private part of the SRK will be the ultimate Root parent key. The result is that the security properties of a wrapped sub-key will be equally to those of the SRK.
The storage keys form the nodes of the key hierarchy, while the signing keys always will be the leaves. That is a storage key can be a parent key for a sub-key, while a signing key can have no sub-keys.
A non-migrateable key can be a parent for either a non-migrateable sub-key or for a migrateable sub-key, while a migrateable key can only be a parent for a migrateable key.

Each TPM object that does not allow "public" access (such as keys) contains a 160-bit shared secret. This shared secret is enveloped within the object itself. The TPM grants use of TPM objects based on the presentation of the matching 160-bits using protocols designed to provide protection of the shared secret. This shared secret is called the AuthData. If an subject presents the AuthData, that subject is granted full use of the object based on the object's capabilities, not a set of rights or permissions of the subject.

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

4.12 Locality on TPM

Locality is a concept that allows various trusted processes on the platform to communicate with the TPM such that the TPM is aware of which trusted process is sending commands. This is implemented using dedicated ranges of LPC addresses as specified in [9].
There are 6 Localities defined numbers 0 – 4 and None.
Locality 4: Trusted hardware component. This is used to establish the Dynamic RTM.
Locality 3: Auxiliary components. Use of this is optional.
Locality 2: Dynamically Launched OS (Dynamic OS) runtime environment.
Locality 1: An environment for use by the Dynamic OS.
Locality 0: The Static RTM, its chain of trust and its environment.
Locality None: This locality is defined for using TPM 1.1 type I/O-mapped addressing.
The TPM behaves as if Locality 0 is selected.

Address range for each locality:

9 TCG_PCClientTPMSpecification_v1-21

System Address	LPC Address	Locality
FED4_0xxxh	0xxxh	0
FED4_1xxxh	1xxxh	1
FED4_2xxxh	2xxxh	2
FED4_3xxxh	3xxxh	3
FED4_4xxxh	4xxxh	4

Each PCR, during manufacturing of the TPM, has the locality level set for two types of operations: reset and extends. The ability to extend the PCR is limited to the specified usage. The usage of the PCR follow this definition:

Index	Usage
0 – 15	Static RTM
16	Debug
17	Locality 4
18	Locality 3
19	Locality 2
20	Locality 1
21	Dynamic OS Controlled
22	Dynamic OS Controlled
23	Application Specific

The TPM enforces locality restrictions on TPM assets such as SEALED blobs restricted by PCR attributes. For example, if a blob is SEALED to PCR 19 (Locality 2), a component executing at Locality 4 (extending PCR 17) cannot UNSEAL the blob. To UNSEAL the blob it would require to extend PCR 19 which is not possible for a component executing at Location 4.

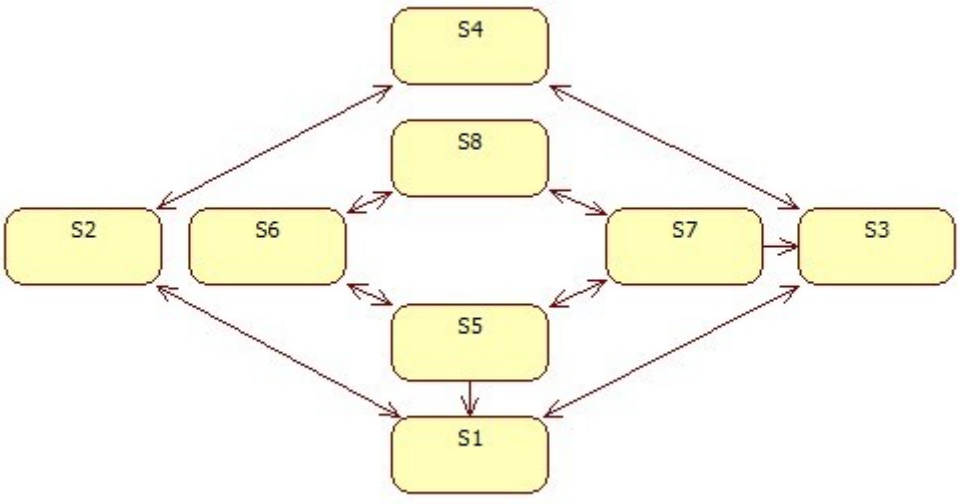
4.13 TPM States

The TPM can be in different activation states that have a direct influence on which commands and actions the TPM is capable of handling. Initially the TPM will be in a state of S8 : Disabled, inactive and unowned. For the TPM to be fully operational, it must be in a state of S1 : Enabled, active and owned.

The TPM can be in the following states:

1. Enabled – Active – Owned
2. Disabled – Active – Owned
3. Enabled – Inactive – Owned
4. Disabled – Inactive – Owned
5. Enabled – Active – Unowned
6. Disabled – Active – Unowned
7. Enabled – Inactive – Unowned
8. Disabled – Inactive – Unowned

This state diagram will show the different state changes the TPM will accept:



The TPM can be returned to state S8 from any state by the commands: TPM_OwnerClear or TPM_ForceClear.

The different states of the TPM have a direct influence on which commands and actions

the TPM is capable of handling:

Enabling a TPM : A disabled TPM is not able to execute commands that use the resources of a TPM.

Activating a TPM : A deactivated TPM is not able to execute commands that use the resources of a TPM. A major difference between deactivated and disabled is that a deactivated TPM CAN execute the TPM_TakeOwnership command and a disabled TPM can not.

Taking TPM Ownership : Control of the TPM revolves around knowledge of the TPM Owner authentication value.

Taking the ownership will result in the following:

- Create the SRK (invalidating any previous SRK).
  Once invalidated all information stored using the SRK is now unavailable. The invalidation does not change the blobs using the SRK rather there is no way to decrypt the blobs after invalidation of the SRK. This also invalidates any child keys that are below the SRK in the key hierarchy.
- Create tpmProof (invalidating any previous tpmProof).
  tpmProof is a value that provides the uniqueness to values stored off of the TPM. By invalidating tpmProof all off TPM blobs will no longer load on the TPM.
- Create the TPM Owner authentication value (invalidating any previous value).
  With the previous authentication value invalidated there are no TPM Owner authenticated commands that will execute with the previous authentication value specified.

5 Analyzes and Results

5.1 Tamper free installation

Let S be an entity that acts as a secure storage location for the protected software. Furthermore S will act as a distribution point for the protected software, in such a way that S must guarantee that it will not reveal the protected software in any other forms than as a protected software package as described below. The distribution can take any form (eg. download over the Internet, using a media like CD or DVD). As the security constraints are placed on the software package, there are no security requirements that the transport protocol used for shipping the package must fulfill. It is assumed that S is part of the Trusted Computing Base. S will be the source from which to retrieve the protected software package prior to initiating the tamper free installation process. It is the task for S to create the protected package as described above.
Thus the responsibilities of S will be:

- Act as a secure storage location
- Create the protected package
- Act as a distribution point

The software package consists of the software to protect against being copied and being hidden from unwanted audience. As the software must not be revealed to an attacker, it is crucial to ensure that it is computationally hard for an attacker to perform the decryption of the software package.
The package must be constructed in a way to ensure it is bound to the platform where the package is to be installed and it must be bound to the Identity of the part of the software package that will perform the installation of the software.
This is where the decryption capabilities of the UNSEAL operation ensures the secrecy of the protected software.
The UNSEAL operation binds the encrypted software package to

- The Identity of the installation software (using the PCR registers) and
- The platform where the software is to be installed (using a non-migrateable private TPM key)

The goal is to ensure it will not be possible for an attacker to decrypt the software package.
This section will focus on how to construct a protected software package that can only be UNSEALED on the specific platform (or more precise, on the platform containing the specific TPM) and only UNSEALED using the trusted installation software.
In the next section "5.2 Tamper free execution", details about how the UNSEAL operation protects this software package from an attacker will be explained. Description of the issue in this section is logical as the same security issues that relate to the protection of the software in the form of a installation package, also relate to the protection of the software after it has been installed.

5.1.1 Detailed description of how to achieve Tamper free installation

In the following sections a detailed description is given on how a secure software package is created in a way that will ensure that the protected software is protected against being copied and being hidden from unwanted audience.

The section will be divided into these two main parts:

- Validating that the TPM is trustworthy using some form of Remote Attestation.
- Collecting required information and constructing a SEALED software package.

5.1.1.1 Validate the TPM using Remote Attestation

As the tamper-free protection scheme rely on the security properties of the TPM, it is crucial that it is in fact a trustworthy TPM we interact with, and not a rogue untrustworthy TPM. Several trust building blocks are used in this attestation:
A 2048-bit RSA key pair called the endorsement key (EK) is a non-migrateable asymmetric key. The private portion of the key exists only in a TPM-shielded location. The EK is created by the TPM manufacturer and is unique for the TPM. A Certificate (Endorsement Credential) with a Trusted Third Party's Signature is used to perform Attestation of the TPM Validity. The Endorsement Credential contains the public part of EK.
In principle, the TPM could use the EK to digitally sign messages. But using the EK to sign messages, would enable the verifier of the signature to uniquely identify the TPM that signed the message, this is unacceptable from a privacy perspective.
The verification operation could be visualized as:

$$\text{TPM} \rightarrow \text{EK}, \text{Sig}_{\text{EK}}(\text{msg}) \rightarrow \text{Verifier}$$

To address this problem the TPM does not use the EK for the operation of signing. Instead, the TPM can generate a random RSA key, called Attestation Identity Key (AIK), which will be used for signing instead of the EK. The TPM can generate an arbitrary number of AIKs. AIKs effectively function as aliases for the EK and, since there can be different AIK keys for each session, then verifiers cannot link the different AIKs as belonging to a particular TPM. This approach will solve the privacy issue that exists in relation to using the EK to sign messages.
The protocol recommended by TCG to use for attestation of TPM trustworthiness is Direct Anonymous Attestation (DAA) as described in [10]. The scheme is provably secure in the random oracle model under the strong RSA and the decisional Diffie-Hellman assumption.
The protocol can be visualized by this simplified diagram:

$$\text{TPM} \rightarrow \text{EK}, \text{Generated DAA} \rightarrow \text{DAA issuer}$$
$$\text{TPM} \leftarrow \text{Sig}_{\text{IS}}(\text{DAA}) \leftarrow$$

$$\text{TPM} \rightarrow \text{AIK}, \text{Sig}_{\text{AIK}}(\text{msg}), \text{proof: Sig}_{\text{IS}}(\text{DAA}) \rightarrow \text{Verifier}$$
$$, \text{proof: Sig}_{\text{DAA}}(\text{AIK, Time, Verifier})$$

10 Ernie Brickell et al., Direct Anonymous Attestation

The TPM Generates a DAA key and send the public part of the DAA key together with EK public Key to the DAA issuer.
The DAA issuer verifies that EK is a valid key and signs the DAA key.
The TPM Generates an AIK key and uses this to sign the message and sends both to the Verifier together with a proof that TPM possesses a DAA that was generated by the TPM and signed by the DAA issuer using a possession proof scheme as described in [11].
The proof that the DAA was generated by this TPM goes like this: As the signature on the AIK is signed using the DAA key, then the TPM must have generated the DAA, as knowledge of the DAA private key part is required to perform this signature on the AIK and the TPM creating the DAA key is the only one knowing the private part of the DAA key. As the signature on the DAA is signed by the DAA issuer, then the TPM that generated the DAA key must be valid, as the DAA issuer is trusted to verify that a TPM is genuine.
There is no linkability between the AIK and the EK, as the EK is not send to the verifier but only to the DAA issuer.
There is only need for the TPM to contact the DAA issuer once, not every time an AIK is generated and used to sign a message, thus the Privacy CA is not a bottle neck.

5.1.1.2 Object-Specific Authorization Protocol (OSAP)

Several TPM commands use the Object-Specific Authorization Protocol (OSAP), so this protocol needs to be described before the description of any TPM communication is commenced.
The OSAP protocol creates an authorized session that forms a secure communication channel between the TPM and the command issuer.
The protocol is designed to prevent an attacker from modifying the commands send to or returned from the TPM, and for authenticating any access to a resource protected by TPM.
The protocol uses rolling nonce's and Hash to protect the messages. For every even nonce generated by the TPM, there is also an odd nonce generated by the caller.

The protocol works as follows and it is more formally described below.

U : Let U be a generic subject that wish to use a resource protected by the TPM.
T : Let T be the TPM.

U → T : TPM_OSAP, keyHandle, nonceOddOSAP
U ← T : authHandle, authLastNonceEven, nonceEvenOSAP

11 Camenisch et.al, Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials

keyHandle : A handle to a key known by U (that is U knows the AuthData for the key).
nonceOddOSAP : A Odd OSAP Session nonce created by U.
authHandle : Session handle created by T. Used throughout the OSAP Session.
authLastNonceEven : A Even nonce created by T.
nonceEvenOSAP : A Even OSAP Session nonce created by T.

Now the OSAP Session is set up. Both U and T will calculate the shared secret.
sharedSecret = HMAC(AuthData, nonceEvenOSAP, nonceOddOSAP)

Let TPM_Command be an example of a TPM command that is executed under the OSAP protocol and takes the arguments inArgOne and inArgTwo. The result of this command will be a returnCode and a result argument outArgOne.
As the OSAP protocol is used to authorize many different commands, this example is kept as generic as possible.

U → T : TPM_Command, inArgOne, inArgTwo, authHandle, nonceOdd, inAuth, continueAuthSession

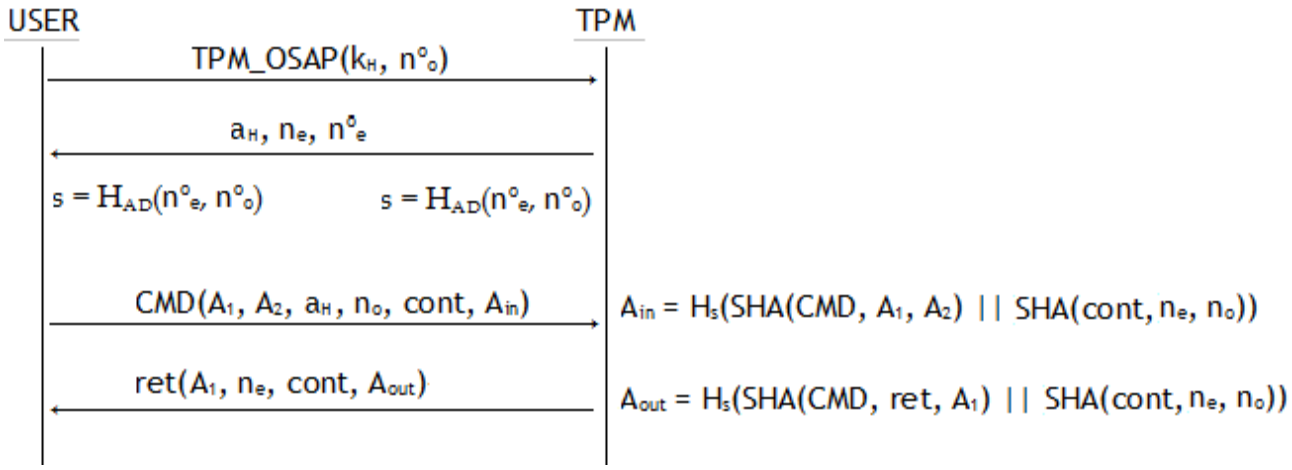
nonceOdd : A Odd nonce created by U.
continueAuthSession : False if the OSAP Session is to terminate.

Both U and T calculate inAuth used to validate the command and input arguments.
inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams).
inParamDigest: SHA1(TPM_Command ordinal, inArgOne, inArgTwo).
inAuthSetupParams : SHA1(continueAuthSession, authLastNonceEven, nonceOdd)

U ← T : returnCode, outArgOne, nonceEven, continueAuthSession, resAuth

nonceEven : A Even nonce created by T. If the Session is to continue, this will be the authLastNonceEven for the next Command to send.

Both U and T calculate resAuth used to validate the result and output arguments.
resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams)
outParamDigest: SHA1(returnCode, TPM_Command ordinal, outArgOne).
outAuthSetupParams: SHA1(continueAuthSession, nonceEven, nonceOdd)



5.1.1.3 Transport Session

TPM commands can be executed under a Transport Session to keep the communication confidential, so this protocol needs to be described before the description of any such TPM communication is commenced.
The protocol is designed to prevent an attacker from modifying the commands send to or returned from the TPM, and for encrypting any data transferred through the transport session (these data would be TPM commands and results).
The protocol uses rolling nonce's and Hash to protect the messages and RSA with an asymmetric 2048 key for encryption. The Key used for encryption must be a non-migrateable storage key. For every even nonce generated by the TPM, there is also an odd nonce generated by the caller.

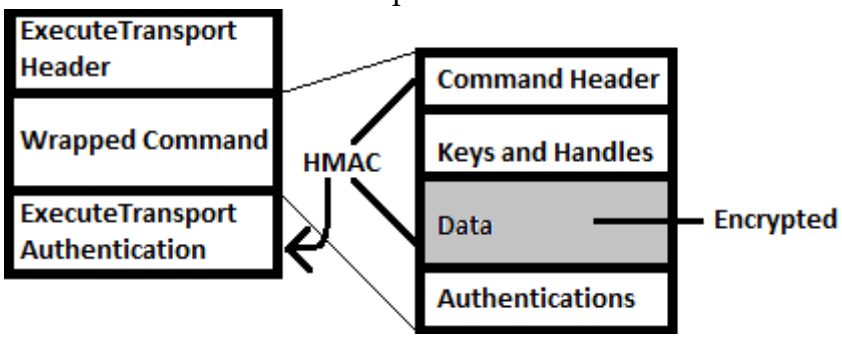
The Transport Session is setup using the TPM_EstablishTransport command.
The only command valid during a Transport session is TPM_ExecuteTransport, which contains an encrypted command to be executed, except for TPM_ReleaseTransportSigned that also is a valid command during a Transport session.
The Transport session will be terminated in case of:

- continueTransSession is false
- TPM_ReleaseTransportSigned command is received
- An erroneous command is received

While the Transport session is in progress, any number of commands can be sent using the TPM_ExecuteTransport command. For example the command sequence of establishing an OSAP protocol session and performing commands under the protection of both the OSAP protocol and the Transport session.
The transport session uses the same rolling nonce protocol that authorization sessions use. This protocol defines two nonces for each command sent to the TPM, nonceOdd provided by the caller and nonceEven generated by the TPM.

The protocol uses HMAC to ensure the integrity of the messages and RSA for encryption to keep the transferred data secret. (Actually the TCG specification specify that RSA encryption support is mandatory, but that the TPM may optionally support other encryption algorithms (such as AES) as long as they are at least as cryptographically strong as a 2048 key RSA encryption).
The HMAC will only include two areas from the wrapped command, the command header information up to the handles, and the data after the handles. The format of all TPM commands is such that all handles are in the data stream prior to the payload or data. After the data comes the authorization information. To enable resource management, the HMAC for TPM_ExecuteTransport only includes the ordinal, the header information and the data. The HMAC does not include any handles or the authorization handles or nonces. The exception is TPM_OwnerReadInternalPub, which uses fixed value key handles that are included in the encryption and HMAC calculation.

The structure of the execute transport command would be like this:



The AIK that was used to perform Remote Attestation of the TPM must be used to sign a message that is sent under the protection of the Transport and OSAP Session. This will ensure that the session is authenticated by the attested TPM that is the designated recipient of the commands. This is performed using the TPM_Quote command. This command takes a key handle and a PCR Register selection as parameters. The resulting value is a Blob signed using the private part of the key specified by the key handle. The Blob contains information about the specified PCR Registers. The PCR informations are not interesting as such and are of no importance, the signature performed on the Blob is what is interesting. As the private part of the AIK symmetrical key is only known by the Remote Attested TPM, then a valid signature will prove that the TPM that forms the other end-point of the secured communication channel is in fact the designated TPM.

An implementation of this protection scheme would most likely use the Storage Root Key (SRK) for encryption of the wrapped command, but the implementation could use any non-migrateable storage key. It is not relevant for the security properties of the protection scheme which specific key is actually used. There are no security concerns with exposure or use of the public portion of the key.
To retrieve the SRK the command TPM_OwnerReadInternalPub would be used. This command can retrieve the public part of either EK or SRK. Using the handle

TPM_KH_SRK will retrieve the SRK key.
To retrieve a non-migrateable storage key that is ancestral to the SRK the command TPM_GetPubKey would be used. This command will retrieve the public part of the key when the handle of the requested key is specified.
The command to retrieve the public part of the key must be executed before the Transport Session is established, as the encryption of the wrapped command will require this information.

This sequence diagram will sketch the commands exchanged under the transport session. The sketching is stripped for details such as rolling nonce values, to avoid that the significant details are lost in numerous nonce parameters:
U : Let U be a generic subject that wish to use a resource protected by the TPM.
T : Let T be the TPM.

First the OSAP Protocol is established:
U → T : TPM_OSAP(Key)
U ← T : ret(authHandle)

The encryption key is retrieved, and the OSAP Session is ended:
U → T : TPM_OwnerReadInternalPub(TMP_KH_SRK, cont)
U ← T : ret(KEY, cont)

The Transport Protocol is established:
U → T : TPM_EstablishTransport(RSA_{SRK}(secret), SRK, cont)
U ← T : ret(transHandle, cont)

All subsequent commands will be wrapped in a Transport Session, this diagram will only show the wrapped command and response, but actually the command syntax will be:
U → T : TPM_ExecuteTransport(transHandle, wrappedCmd, transAuth)
U ← T : ret(wrappedResponce, cont, transAuth)

The OSAP Protocol is established:
U → T : TPM_OSAP(AIK)
U ← T : ret(authHandle)

The TPM_Quote command is sent under both Transport Protocol and OSAP Protocol:
U → T : TPM_Quote(AIK, PCR)
U ← T : ret($Sign_{AIK}$(Blob), PCR)

The OSAP Protocol is established:
U → T : TPM_OSAP(Key)
U ← T : ret(authHandle)

The SEAL command is sent under both Transport and OSAP Protocol:
U → T : TPM_SEAL(PCR, Data, authHandle, cont)
U ← T : ret(sealedData, cont)

5.1.1.4 Create a SEALED package of the protected software

The SEAL operation binds the encrypted software package to the Identity of the installation software (using the PCR registers) and to the platform where the software is to be installed (using a non-migrateable private TPM key).
The TPM uses symmetric encryption and decryption using a public / private symmetric key pair. There are no security concerns with exposure or use of the public portion of the key.
Using the accessible public key from the specified TPM to encrypt the protected software package will ensure that the package can only be decrypted by the same TPM by usage of the private key, that are only known by that specific TPM. The private portion of the SRK exists only in a TPM-shielded location. The private portion of any non-migrateable child key will be wrapped and has the same security properties as the SRK.

In order to be able to construct a protected software package that can only be UNSEALed using the specific TPM, the following information must be procured:

- The public part of a non-migrateable storage key of the specific TPM.
- The content of the PCR register that measures the protected software.

The following sections will describe in more details how these informations are obtained.

Create a SEALED package performing a SHA-1 hash of the protected software to create a digest and perform an RSA encryption of the protected software using the public key.

Performing encryption using a symmetric key encryption method like RSA is a factor of several magnitudes slower than using a secret key encryption like AES.
A practical implementation that is to encrypt a potentially large amount of data like the protected software would be, could be impaired due to low performance in connection with performing a symmetric encryption scheme to a large amount of data.
A more practical usable method would be to perform a secret key encryption on the protected software, and then perform the symmetric encryption on the secret key, which is much smaller in size (in the magnitude of 160 bits or so), and thus not as performance expensive to encrypt using the symmetrical encryption method.
Such a choice of implementation would not reduce the cryptographic requirements for the symmetric encryption performed in the protected software package, as a leak of the secret key would be as crucial for the confidentiality of the protected software, as a leak of the clear-text protected software would have been. As the actual implementation details is out of scope for this report, then this additional encryption step likely included in a actual implementation will not be addressed further.

To encrypt the protected software we need to use the public part of a non-migrateable storage key of the specific TPM. Encryption with the public part of the symmetrical key pair will ensure that only the TPM can decrypt the package using the private part of the symmetrical key pair. Thus for an attacker to retrieve the clear-text software, it will require the knowledge of the private key which exists only in a TPM-shielded location, or it will require breaking the encryption which is believed to be computationally hard.
To bind the protected software to the trusted software we need the expected content of the PCR register that measures the trusted software. Binding the software ensures that only software with a PCR register measurement that match the trusted software will be able to decrypt the package. Thus for an attacker to invoke the unseal operation , it will require finding a collision for the hash value that forms the software measurement which is believed to be computationally hard.
It is crucial that the command to create the sealed package is executed under the protection of the OSAP Session and the Transport Session, and that a message send under these protocols is signed by the Attestation Identity Key (AIK) used for remote attestation to ensure the coupling between the attestation operation and the package sealing operation. The TPM that creates the sealed package must be no other than the TPM that performed the attestation.
These protocols create a secure channel with the trusted TPM as an end-point. This extends the TCB to include the commands executed in this secure channel, thus comply with the restriction of not revealing the protected software outside the TCB in any other form than as a protected software package.
An implementation of this protection scheme would most likely use the Storage Root Key (SRK) for encryption, but the actual implementation could use any non-migrateable storage key. It is not relevant for the security of the protection scheme which specific key is actually used.
The tpmProof is a TPM secret that exists only in a TPM-shielded location, and never leaves the TPM in clear-text. The TPM use this value to validate that a external data structure is actually created by this specific TPM.
Here is sketched an overview of which cryptographic building blocks the SEAL command uses to create the secure software package:
Let T be the software part that needs to be authenticated (The trusted software).
Let P be the software part that needs to be encrypted (The protected software).
Let PCR# be the number of the PCR Register to use for measurement of T
Let PUBKEY be the non-migrateable storage key used for encryption

Create S1 a TPM_STORED_DATA with

- sealInfo.digestAtRelease = hash(T) and
- sealInfo.pcrSelection = PCR#

Create S2 a TPM_SEALED_DATA with

- storedDigest = hash(S1) and
- data = P

$S1.encData = encrypt_{PUBKEY}(S2)$

On a more detailed level the following pseudo-code describes the steps involved in the SEAL operation:

Let the arguments for the SEAL command be represented by:

```
TPM_KEY_HANDLE PUBKEY : Handle of a loaded key that can perform seal operations.
TPM_ENCAUTH encAuth  : The encrypted AuthData for the sealed data.
UINT32 pcrInfoSize   : The size of the pcrInfo parameter.
TPM_PCR_INFO_LONG pcrInfo : The PCR information.
UINT32 inDataSize    : The size of the P parameter
BYTE[ ] inData=P : The data to be sealed to the platform and any specified PCRs

X1 = pcrInfo
S1 = new TPM_STORED_DATA
S1.sealInfoSize = pcrInfoSize
S1.sealInfo.pcrSelection = pcrInfo.pcrSelection
h1 = composite hash of the PCR
S1.sealInfo.digestAtCreation = h1
S1.sealInfo.digestAtRelease = pcrInfo.digestAtRelease
a1 = decrypt_authHandle( encAuth )
S2 = new TPM_SEALED_DATA
S2.payload = TPM_PT_SEAL
S2.tpmProof = TPM_PERMANENT_DATA.tpmProof
h3 = SHA-1( S1 )
S2.storedDigest = h3
S2.authData = a1
S2.dataSize = inDataSize
S2.data = inData
s3 = encrypt_PUBKEY( S2 )
S1.encDataSize = size of s3
S1.encData = s3
Return S1
```

5.1.2 Attack methods on Tamper free installation

5.1.2.1 Man In The Middle (MITM) Attack

In a MITM attack the attacker makes independent connections with the two end-points of the communication and relays messages between them, making them believe that they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker. In this report it is assumed that the attacker is able to intercept all messages going between the two end-points and inject new messages. That is the attacker have complete control of the communication line.

Peek protected software
If the attacker could peek the protected software from the message stream, then the attacker would have gained access to the secret software thus breaking the proposed protection mechanism.
The protected software is transferred as a parameter to the SEAL command. The SEAL command is performed under an Encrypted Transport Session. The Transport Session perform encryption of all parameters to wrapped commands (such as the protected software that is a parameter to the wrapped SEAL command).
To gain access to the protected software would require the attacker to break the RSA encryption which is computationally hard, or to gain knowledge of the private part of the non-migrateable storage key that was used by the transport session to perform the encryption of the wrapped parameters. The private portion of the key exists only in a TPM-shielded location. Should the private portion of the key leave the TPM, then it would be as a wrapped key. The security properties of a wrapped sub-key will be equally to those of the parent SRK.

Modify keys
If the attacker could change the key handle of the encryption key used in the SEAL command to a key handle of the attackers choice, then the encryption key could be replaced with a key having other security properties. This could be a migrateable key, that could be moved away from the designated TPM, thus breaking the binding to the TPM, which is an important security property of the proposed protection mechanism.
The handle of the encryption key is transferred as a parameter to the SEAL command. The SEAL command is performed under an Encrypted Transport Session, but the Transport Session does not encrypt key handle parameters of the wrapped commands. The SEAL command is also under the OSAP Protocol protection. The OSAP Protocol protects the command parameters from modification by including a HMAC of a digest of the parameters using a shared secret. The digest is calculated as a SHA-1 of the command parameters.
The attacker do not know the shared secret that is required to generate the HMAC, so to change a key handle would require the attacker to forge the HMAC. Forging a HMAC is

computationally hard.

Modify PCR informations
If the attacker could change the PCR requirements to any requirements of the attackers choice, then the attacker could insert PCR requirements matching the measurement of the attackers software. It would then be possible for the attacker to perform the UNSEAL command from his own software and retrieve the secret software thus breaking the proposed protection mechanism.
The protected software is transferred as a parameter to the SEAL command. The SEAL command is performed under an Encrypted Transport Session. The Transport Session performs encryption of all parameters to wrapped commands (such as the protected software that is a parameter to the wrapped SEAL command).
To change the PCR requirements would require the attacker to break the Transport Session encryption which is shown secure earlier in this section.

5.1.2.2 Rogue TPM Attack

A Rogue TPM will not necessarily comply with the security properties required of a TPM. Entrusting a untrustworthy TPM with the secret software might result in leaking these security sensitive informations to the attacker.
If the attacker could redirect the communication to a rogue TPM, after a successful Remote Attestation of a genuine TPM, then the attacker could retrieve the protected software from the rogue TPM after the communication has successfully completed.
The operation of creating a sealed package is performed under a Transport Session. All commands sent under the session include a reference to the transport session handle, thus each command must be executed under this session and by the designated TPM with which the transport session is initiated. Before the secret software is sent to the TPM, a Quote command is performed where the TPM is requested to sign the message with the AIK key used in the Remote Attestation of the TPM. To perform the Quote would require knowledge of the private part of the AIK key that was used for Remote Attestation. The private portion of the key exists only in a TPM-shielded location in the Remote Attested TPM. Breaking the symmetrical key signature is computationally hard.

5.1.2.3 Brute-Force Attack on TPM

A Brute-Force attack is performed when the attacker systematically attempts to use every possible value of the key until the correct key value is found. If the value of the key is a uniformly random number then the Brute-Force search might need to cover the entire key value space or an average of half the value space, while some heuristics may be used to reduce the possible result space according to the class of key value that is searched for.
If the attacker could perform a Brute-Force attack on the TPM to retrieve the 160 bits AuthData used to authorize the commands performed on any non-public objects, then the

attacker would be able to perform any command on this object that would require authorization to be performed.
The entropy of the AuthData may be much smaller than 160 bit depending on the actual implementation of the AuthData calculation. If for example the attacker knows that the implementation of the AuthData calculation is a Hash of a user supplied password, it would be possible for the attacker to calculate a Hash of often used passwords, and thus have a high chance of guessing the correct AuthData value in relatively few attempts.
The TPM is required to implement the following protection mechanism against Brute-Force attacks:
The TPM keeps a count of failed authorization attempts. When the count exceeds the threshold of failed authorizations, the TPM locks up and does not respond to any requests for a time out period. The time out period doubles each time the count exceeds the threshold. This protection scheme makes a Brute-Force attack practically impossible, as the logarithmic increase in the length of the time out period would thwart any attack based on numerous failing authorization attempts.

5.1.2.4 Replay Attack

A replay attack occurs when an attacker records a stream of legitimate messages between two parties and replays the stream to one or more of the parties.
An attacker is unable to respond correctly to the Quote challenge, as the ability to sign the message correctly would require knowledge of the private part of the AIK. One could imagine a replay scenario where an attacker by listening and resending parts of the communication between the sender and the TPM could bypass the Quote challenge.
If the attacker recorded the messages the sender would transmit to the legitimate TPM including the answer to the Quote challenge. Then the attacker injected an erroneous reply to the sender, to make the sender abort the process and restart the communication. The attacker captures the communication channel and replays parts of the captured communication to get past the point in the process where the Quote challenge is replied. After this point in the process the protected software is sent to the attacker with an encryption that is based on key information provided by the attacker.
The OSAP and Transport Protocols uses Nonce's to protect against replay attacks. A Nonce (Number Once) is an arbitrary (or pseudo-random) value that is used only once. If a nonce value appear a second time in the communication stream, then the message is being replayed, and the communication is suffering from a replay attack. If the attacker do not change the nonce values in the message, then the receiver of the message can easily detect that the message is being resend as part of a replay attack. If the attacker do change the nonce value, then the parameter authentication digest will not match. This digest is contained in a HMAC using SHA-1 of the transmitted nonce's and a shared secret. An attacker that wish to change the nonce undetected must forge the HMAC, which is computationally hard.

5.2 Tamper free execution

The tamper free software to execute can be separated into two distinct parts:

- The trusted software part.
- The protected software part.

The trusted software is the part of the software that decrypt and execute the protected software (it can be seen as a loader of the protected software).
As the protected software must not be revealed to an attacker, it is crucial to ensure that only the trusted software will be able to perform the decryption.
The Identity of the trusted software must be indisputable before access to the protected software is granted. This is where the integrity measurement mechanism of the PCR registers ensures that access to the protected software is possible only for the trusted software.
The protected software is the part of the software to protect against being copied and the part of software being hidden from unwanted audience.
As the protected software must not be revealed to an attacker, it is crucial to ensure that it is computationally hard for an attacker to perform the decryption of the protected software. This is where the encryption capabilities of the SEAL operation ensures the safety of the protected software. As the SEAL operation binds the secret to the Identity of the trusted software (using the PCR registers) and to the current platform (using a non-migrateable private TPM key), it will not be possible for an attacker to decrypt the protected software.
It is also crucial that an attacker will not be able to peek at the protected software, while it is located in memory in an unencrypted state. This is where the Protected Execution mechanism of the SENTER / SKINIT operation ensures that the protected software is located in a protected memory location that is inaccessible to an attacker.

Use **SENTER** on an Intel Family CPU or **SKINIT** on an AMD Family CPU, to enable Execution in Protected Memory to avoid revealing the protected software to attackers, and to ensure the Identity of the trusted software, and that both parts of the software is unmodified.
As the protected software must not be revealed to an attacker, it is crucial that the protected software can only be decrypted under the Protected Execution. As the protected execution mechanism validates the Identity of the trusted software (using the PCR registers) before the Protected Execution is initiated, it will not be possible for an attacker to perform the UNSEAL operation without execution under Protected Execution (as the specified PCR register will only contain the required value if a piece of software with the trusted software's Identity is running under Protected Execution).

Use the T**PM_UNSEAL**[3] operation to decrypt the protected software when the following conditions are met:

- only on this platform, and

3 TPM Main-Part 3 Commands_v1.2

- by software with the specified Identity, and
- only while the current platform configuration (as defined by the named PCR contents) is the one named as qualified to decrypt it.

As the protected software must not be decrypted by any software other that the trusted software, and not anywhere else than on the specified platform, it is crucial that the decryption can only be performed when the appropriate Identity measurement (ensured by the specified PCR register) and the correct non-migrateable private key (ensured by the key protection mechanism of the TPM) are available.

Using the same PCR register for Protected Execution and for TPM_Unseal (PCR 17) will ensure that only the designated trusted software with a specified Identity can access the protected software.

5.2.1 Detailed description of how to achieve Tamper free execution

In the following sections a detailed description of how the security mechanisms provided by the TPM_SEAL operator and the Intel SENTER operator will apply protection.

5.2.1.1 SENTER on Intel Family CPU

SENTER will enable Execution in Protected Memory to avoid revealing the protected software to attackers, and will ensure the Identity of the trusted software.
In the following sections a detailed description of the security mechanisms used by SENTER will be described.

Measured Identity of Virtual Machine Monitor (MVMM)

The Virtual Machine Monitor (VMM) is a method to sandbox and isolate the individual Virtual Machine's from each other and from the physical hardware. This mechanism is used by the SENTER operation to enable the Protected Execution of the protected software.

Measured Virtual Machine Monitor (MVMM) has a unique Identity specified by the measurement of the VMM calculated using the SHA-1 hashing algorithm. During initialization and load of the MVMM the TPM PCR registers used by the chipset to ensure that the MVMM is unmodified, and thus not tampered with.

Two obvious reasons for the MVMM to assume a different Identity could be:

- The VMM wishes to masquerade as some other trusted entity.
- Some attack corrupted a trusted VMM and the attacker wants to hide.

The Identity of VMM is measured using Platform Configuration Register (PCR), a 160-bit storage location for discrete integrity measurements. All PCR registers are shielded locations and are inside of the TPM. The PCR is designed to hold the measurement of an

unlimited number of measured objects in the register. It does this by extending the contents of the PCR register by using a cryptographic hash and hashing all updates to a PCR. The pseudo code for this is:

PCRi New = HASH (PCRi Old value || value to add)

There are two salient properties of cryptographic hash that relate to PCR construction. Ordering – meaning updates to PCRs are not commutative. For example, measuring (A then B) is not the same as measuring (B then A). The other hash property is one-way-ness. This property means it should be computationally infeasible for an attacker to determine the input message given a PCR value. Furthermore, subsequent updates to a PCR cannot be determined without knowledge of the previous PCR values or all previous input messages provided to a PCR register since the last reset.

The HASH function used is HMAC based on SHA-1, the HASH construct is:

H(K XOR opad, H(K XOR ipad, text))

where

H = the SHA-1 hash operation
K = the key or the AuthData
XOR = the xor operation
opad = the byte 0x5C repeated B times
B = the block length
ipad = the byte 0x36 repeated B times
text = the message information to hash.

Measurement of the MVMM is performed by the processor that starts the GETSEC [SENTER] sequence. The measurement is performed by executing a TPM_Extend command to PCR Register 17. PCR Register 17 has locality 4, which is addressed to 0xFED4_4xxx. The Input Output Controller Hub (ICH) blocks all access to 0xFED4_4xxx unless the CPU indicates that the GETSEC [SENTER] microcode is executing. This construction ensures that PCR Register 17 can be extended with nothing else than the measurement of the MVMM during the execution of GETSEC [SENTER].

Protected Memory Page (Paging, NoDMA and SMI)

To achieve a copy protection of the protected software and a protection of Intellectual Property, the code must be kept hidden from unwanted audience. To reach this goal the software will be executed in Protected Memory Pages by using the SENTER instruction.

Four mechanisms for a software process to gain access to a physical memory page are:

- The software can request access to a physical memory page by referencing the virtual page address. The request to access the corresponding physical page is routed to the memory controller.
- A device can access a physical memory page. These Direct Memory Access (DMA)

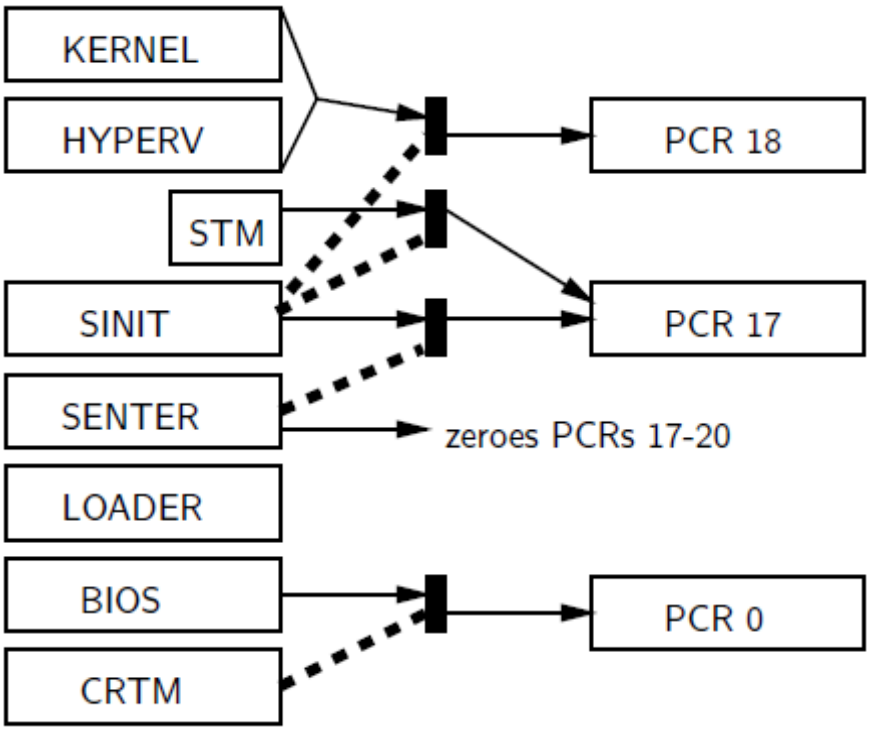
require no CPU control; hence, the paging mechanism is not in use. Software can configure the device to which physical page to use for DMA access.

- Display adapters use a special case of DMA. Special tables, Trusted Graphics Translation Table (TGTT), are made available to the display adapter to achieve quick and efficient access to the display buffers.
- System Management Interrupts (SMI) result in a mode switch in the CPU and will bypass the paging mechanism.

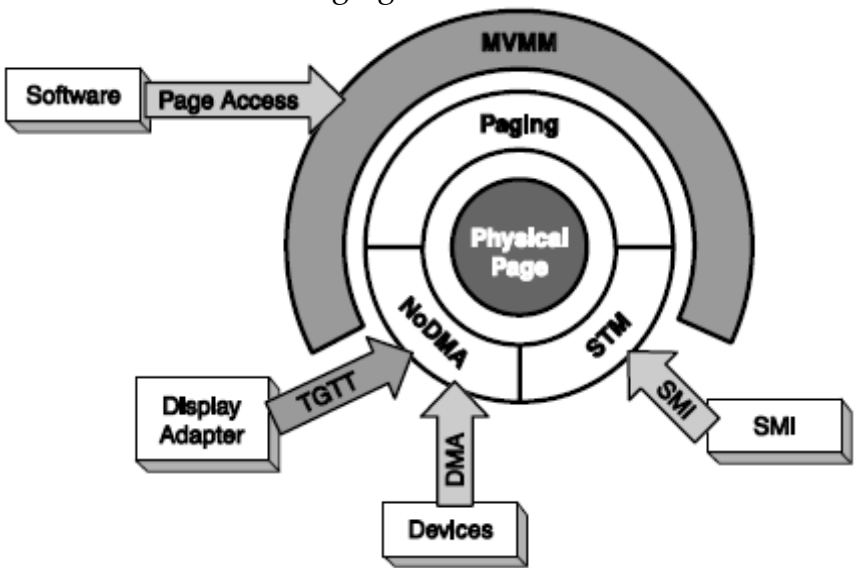
All four of these physical memory page access methods must be taken into account.
Three different protection mechanisms is used to control the memory access.
These components are:

- Paging Mechanism. The paging mechanism is controlled by manipulating CR3, the register that contains the base address of the page directory. Any entity that can control CR3 has the ability to control the access to physical memory through paging. The VMM will only see a virtual CR3. With a validated way of changing the CR3 and the page tables, it is ensured that only access is granted to a proscribed set of physical memory pages.
- NoDMA. While the DMA access will bypass the MVMM and the paging mechanism, the MVMM does control the NoDMA table. Any attempt to touch other physical pages using a DMA access requires the NoDMA table to allow the physical page access.
- SMI Transfer Module (STM). The STM and MVMM negotiate the correct policies for what the System Management Interrupt (SMI) handling code should and should not do, and then the MVMM allows the STM to handle all SMI events. The STM is part of the trust chain for the platform to measure the code and then report the measurement to the TPM.

A block-diagram of STM in the trust chain is presented here:



The relationship between the four memory access methods, and the three protection mechanisms are shown in the following figure:



SENTER Instruction (using AC Module SINIT)

The SENTER Instruction enable Execution in Protected Memory to avoid revealing the software to attackers, and ensures the Identity of the software, thus the software is unmodified.

The Secure Launch involve the following steps:

- Rendezvous all physical and logical processors.

- Protect against outside events. Mask the INIT, A20M, NMI and SMI interrupts.
- Load, verify and execute SINIT, an Authenticated Chipset Module (ACM).
- Protect and measure the MVMM.
- Store the MVMM measurement in the TPM.
- Launch the MVMM, and enable Interrupts.

Measurement, Initialization and Launch of the MVMM is performed by the Authenticated Chipset Module (ACM) named SINIT. The chipset manufacturer is the entity that must vouch for an ACM module and perform a digital signature on the module. The ACM Header contains a PKCS RSA Signature that can validate the module. The ACM is loaded into a special area of internal RAM within the processor referred to as the authenticated code execution area. This enables the execution of ACM to occur in isolation with respect to the contents of external memory and activity on the external processor bus.

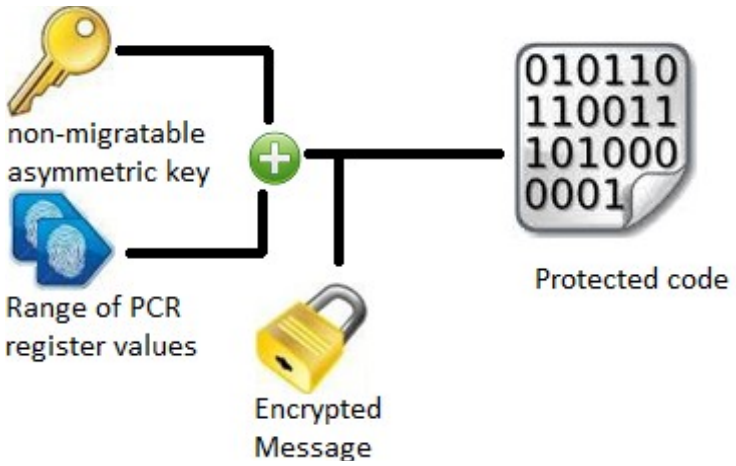
In the actual implementation the SINIT module is split up in two modules: SINIT and SCLEAN that are both ACM's.

5.2.1.2 TPM_Unseal on TPM

UNSEAL decrypt the protected software only on this platform (ensured by a unique TPM specific non-migrateable private key) and by software with the specified Identity (ensured by the Identity measurement of the PCR register).
In the following sections a detailed description of the security mechanisms used by UNSEAL will be described.

UNSEAL Operation (Using PCR and SRK)

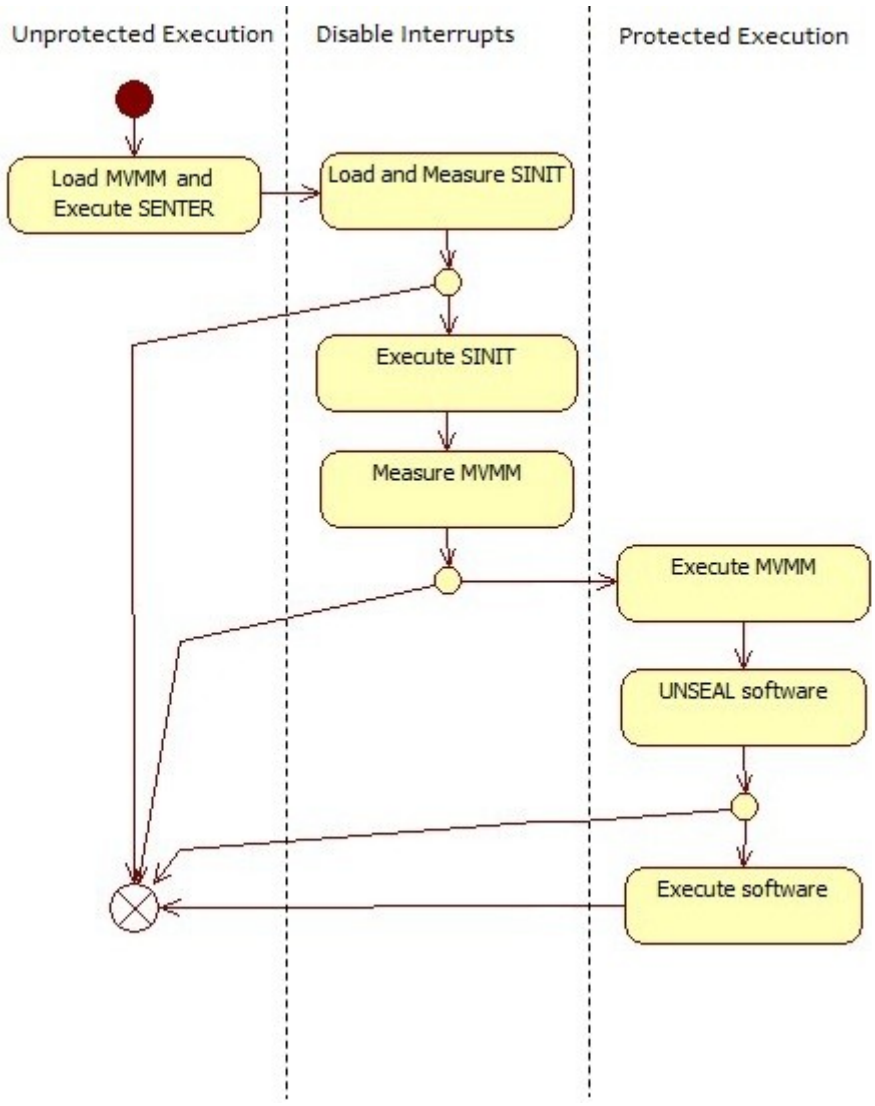
Sealed messages are bound to a specified set of platform metrics. Platform metrics specify the platform configuration state that must exist before decryption will be allowed. Sealing associates the encrypted message with a set of PCR register values and a non-migrateable asymmetric key. A sealed message is created by selecting a range of PCR register values and asymmetrically encrypting the PCR values plus the symmetric key used to encrypt the message. The TPM with the asymmetric decryption key may only decrypt the symmetric key when the platform configuration matches the PCR register values specified by the sender.



The PCR Registers associated with the encrypted message is the Identity of the trusted code used to decrypt the protected software (MVMM), and may also be the metrics that specify the current platform configuration state. The value in these PCR Registers ensures that the protected software can only be decrypted by the trusted software, and it ensures that the decryption can only be performed on a platform in a trustworthy configuration state according to the chain of trust ensured by the Root of Trust.

The non-migrateable asymmetric key associated with the encrypted message may be the Storage Root Key (SRK). The SRK like the Endorsement Key (EK) are embedded in the TPM. These keys cannot be removed from the TPM. The private portion of these keys exists only in a TPM-shielded location. The value of this private key ensures that the protected software can only be decrypted by this specific TPM.

This diagram will show the overall sequence of the processes:



5.2.1.3 Exit Protected Execution

After completion of execution of the protected software, the execution in Protected Memory needs to be terminated securely to avoid revealing the protected software to attackers after the protected execution has terminated.
The SEXIT operation ensures that no secrets is exposed from the protected partition after the protected environment is shutdown. The SEXIT operation can only be called from within the MVMM.

The Secure Launch Shutdown involve the following steps:

- Rendezvous all physical and logical processors.
- Clear the MVMM measurement from the PCR in TPM.
- Shutdown the MVMM and scrub the contents of the protected partition from system memory.

5.2.2 Attack methods on Tamper free execution

5.2.2.1 Read Protected software from SEALED package

If the attacker could peek the protected software from the SEALED package, then the attacker would have gained access to the secret software thus breaking the proposed protection mechanism.
The protected software in the SEALED package is encrypted with RSA using a non-migrateable storage key. To gain access to the protected software would require the attacker to break the RSA encryption which is computationally hard, or to gain knowledge of the private part of the non-migrateable storage key that was used to perform the encryption. The private portion of the key exists only in a TPM shielded location. Should the private portion of the key leave the TPM, then it would be as a wrapped key. The security properties of a wrapped sub-key will be equally to those of the parent SRK.

5.2.2.2 UNSEAL on Other Platform

If the attacker could perform the UNSEAL command on another platform than the platform that contains the designated TPM, this would mean breaking the binding to the TPM, which is an important security property of the proposed protection mechanism.
The protected software in the SEALED package is encrypted with RSA using a non-migrateable storage key. To perform the UNSEAL command on another TPM, would require the attacker to gain knowledge of the private part of the non-migrateable storage key that was used to perform the encryption.

5.2.2.3 UNSEAL from Other Software

If the attacker could perform the UNSEAL command from his own software, then the attacker would have full access to the clear-text of the protected software. This would break the security property of the proposed protection mechanism that is specifying that the protected software should be kept secret from unwanted audience at any time.
The protected software is SEALED with the PCR Register contents that match the measurement of the trusted software. To perform the UNSEAL command, a value matching HMAC of the SHA-1 digest of the trusted software must be found in the specified PCR Register. To perform a UNSEAL command would require the attacker to either create software with a measurement that would match the measurement of the trusted software or forge the HMAC. Forging a HMAC is computationally hard.

5.2.2.4 UNSEAL without Protected Execution

If the attacker could perform the UNSEAL command from the unmodified trusted software, but manipulate the execution of the program in such a way that initial step that

enable the Protected Execution is skipped, and only the subsequent UNSEAL command is performed, thus executing the trusted program in a vulnerable unprotected state. This would give the attacker access to peek the memory where the clear-text of the protected software is located after the UNSEAL command is successfully completed. This would break the security property of the proposed protection mechanism that is specifying that the protected software should be kept secret from unwanted audience at any time.
The protected software is SEALED with the PCR Register that can only be used by the SENTER operation to measure the Identity of the loaded MVMM. Without the Protected Execution being in progress the specified PCR Register will not contain the measurement of the trusted software. When the specified content is not found in the specified PCR Register, the the UNSEAL command will fail.

5.2.2.5 Memory Access

The Protected Execution mechanism enforced by SENTER, ensures that access to the protected memory containing the protected software is prohibited.
To access the protected memory pages, the Protected Execution mechanism must be circumvent.

Memory Page Access from an Application

An application can request access to a physical memory page by referencing the corresponding virtual page address. The paging mechanism is controlled by manipulating the CR3 register. The Protected Execution mechanism ensures control with the CR3 register, so access only is granted to a proscribed set of physical memory pages.
This ensures that the Protected Memory is inaccessible to any application other than the software running under Protected Execution.

DMA / TGTT Access from a Device or Display Adapter

A device or a display adapter can use DMA to access a physical memory page. The NoDMA Table allow access to the physical page. The Protected Execution ensures control with the NoDMA Table, so access only is granted to a proscribed set of physical memory pages.
This ensures that the Protected Memory is inaccessible to any device or display adapter.

Memory Page Access from SMI

The SMI Transfer Module (STM) is a sandbox for execution System Management Interrupts (SMI) code according to the policies negotiated between the STM and the MVMM. The STM is part of the trust chain for the platform and is measured and validated by SINIT. The STM ensures that SMI is only granted access to a proscribed set of physical memory pages.

This ensures that the Protected Memory is inaccessible to any SMI code.

5.2.2.6 Attack SINIT

Initialization performed by the SENTER Operation is done using SINIT. A way to circumvent the Protected Execution mechanism is by modifying the SINIT software to cripple the SENTER protection. SINIT is a software file located on the Computer hard-drive, and is thus vulnerable to modification. As a Authenticated Chipset Module (ACM) the SINIT software contains a digital signature where the chipset manufacturer vouches for the module.

Modifying Authenticated Chipset Module (ACM)

One way an attacker could perform an undetected modification of the ACM, would be to modify the ACM so the HMAC SHA-1 hash would be similar to the original module. Forging a HMAC is computationally hard as shown earlier in this report.

Another way would be to break the Digital Signature to be able to sign any arbitrary software as the original ACM. This would require breaking the RSA signature scheme. Breaking RSA is computationally hard as shown earlier in this report.

The conclusion must be that it would not be possible for an attacker to perform an undetected modification of the SINIT software in order to cripple the Protected Execution mechanism.

Security Holes in SINIT

The ACM signature of the SINIT will only guarantee that the software is unmodified and vouched for by the chipset manufacturer.

It still requires you to have trust in the chipset manufacturer, and their ability to produce reliable software that performs the desired functionality without errors or security vulnerabilities.

As shown by [8] the Intel SINIT code was vulnerable to a classic buffer overrun exploit, where a modification of the unprotected DMAR ACPI table can cause an overflow to the buffer located on the TXT heap, and allow the attacker to execute arbitrary code in the SINIT module. Intel replied with an update of the vulnerable SINIT module to avoid the exploit on December 5th, 2011.

The conclusion must be that even as the signature mechanism of ACM ensures that SINIT

8 Wojtczuk & Rutkowska, November 2011, Attacking Intel TXT® via SINIT code execution hijacking

is unmodified as vouched for by the chipset manufacturer, then the actual implementation of the SINIT module still rely on the trust in the chipset manufacturer's ability to produce reliable and secure software.

The fact that Intel SINIT is in wide use, and that someone have taken the time to analyze the code for security exploits, this increases the probability that it is actually secure.

5.2.2.7 Attack TPM device

In practice different manufacturers of TPM devices may implement TPM differently: they may exploit the flexibility that the TPM specification itself provides, or they may deviate from the specification by inappropriate design that might lead to security vulnerabilities.

If a certified TPM device is implemented in such a way that it deviates from the TPM specifications as published by the TCG, then this might influence the trustworthiness of the TPM, and allow an attacker to implement security exploits that reveal the secrets entrusted to the TPM.

An example of such a deviation from the specifications that could lead to a security vulnerability could be neglecting the required freshness of key handles.
The TPM 1.2 specification requires certain quality standards for the randomness of handles. Those include the key handles returned during a TPM_LoadKey as well as any session handle created with TPM_OIAP or TPM_OSAP. The main problem relies in any missing freshness of the received handle. If an attacker is able to evict a key out of the TPM, the associated key handle is deleted inside the TPM and any further reference to this key handle (e.g., through a key operation) is impossible. By ignoring the freshness requirement, an attacker might figure out how the TPM generates and associates session and key handles and can therefore perform an attack using this information. If for example the attacker knows that a freed key handle will be immediately assigned to the next key loaded into the TPM. It is possible to evict a certain key, create an own key and load that key into the TPM. Since the TCG specifications do not consider performance issues, an application executing a TPM command has no estimation about the processing time. Therefore, this attack can be extended to a Man In The Middle attack, where e.g. a malicious device driver may intercept the communication with the TPM, analyze the command and exchange the key, that the key handle refers to. An application trying to perform a key operation (e.g. sealing) might not even realize that the key has been exchanged. Therefore, data to be sealed might be encrypted with a key not even in the possession of the legitimate application, but only known by the malicious exploit software.

The fact that any legitimate TPM device must be evaluated by an independent 3rd party Conformance Entity that is to validate the conformance of the device according to the ISO-15408 Common Criteria standard for evaluating computing systems, this increases the probability that a legitimate TPM device is actually conforming to the specifications.

5.2.2.8 Attack by Reset

When the Reset pin is set, each CPU will perform a hardware initialization. All registers are set to a known state, and all state information is cleared. System memory will retain information during a system reset. The above described properties exists for a power-down immediately followed by a power-up. Such a power cycling will be treated in a similar manor as a reset event in this section.

If the attacker performs a Reset at the time when the secret software is executing under the Protected Execution, then the secret software would exist in clear-text in system memory, while the protection of the Protected Execution would be removed as a result of the hardware initialization of the CPU that is caused by the Reset. After the BIOS startup process is completed, the attacker would have access to the secret software located in clear-text in system memory.

To protect against this type of attack, the system (or to be more precise the Intel Trusted Execution Technology (TXT) platform) needs a method to mitigate the exposure of any system memory that could contain secrets to an attacker after a Reset.
To implement such a method, the system needs to:

- Determine if a Protected Execution was in progress before the Reset occurred.
- Perform a scrub of sensitive system memory before allowing access to it.

Below is explained how TXT achieves these two countermeasures.

Method of determining if Protected Execution was in progress.

A small battery exists called the 'power well' that persists ICH (Input Output Controller Hub) informations as date, time and other such values independent of the availability of the main PC power or laptop battery. The ICH maintains a LT.HASSECRETS flag in the power well. This flag is set when performing the SENTER operation and is cleared when performing the SEXIT operation. The flag will thus be set during the Protected Execution and will remain set if the protected execution is aborted due to a Reset. The ICH has a sensor that is set if the battery power is not sufficient to hold the informations in the power well. Should the sensor be set caused by a battery malfunction or deliberate removal, then the ICH will treat this equally to the LT.HASSECRETS flag being set.

Performing a scrub of system memory before allowing access to it.

After a reset the MCH (Memory Controller Hub) blocks all access to system memory. If the ICH determines that no Protected Execution was in progress before the Reset occurred, then the MCH will unblock the access to system memory.
Should the LT.HASSECRETS flag be set, then the BIOS must initiate the scrub of system memory. An Authenticated Chipset Module (ACM) named SCLEAN performs the task of scrubbing the system memory. The SCLEAN module has the same security properties as the SINIT module, and the security arguments stated earlier in this report regarding the SINIT module also apply to the SCLEAN module. The SCLEAN module has the ability to clear the LT.HASSECRETS flag. Thus after the scrub of system memory is performed by

SCLEAN, then the MCH will unblock the access to system memory.

5.2.2.9 Attack by Change of Power Level

According to the ACPI (Advanced Configuration and Power Interface) specification there is five different power states:

- S0 - On
- S1 – Standby : The monitor is off and the CPU is in a sleeping state, but no system context (neither CPU or chipset) is lost.
- (S2) – Unused power state. Should be similar to S3 according to ACPI specifications.
- S3 – Suspend-To-RAM : All system context is lost (both CPU and chipset) except system memory.
- S4 – Suspend-To-Disk : All system memory and system context are flushed to a hibernation file on disk and the system is powered off. At resume the system memory and context are reestablished and normal operation is resumed.
- S5 - Off

If the attacker causes a change in power state to S3 or S4 at the time when the secret software is executing under the Protected Execution, then the protection of the secret software would be compromised.

If changing to power state S3 then the secret software would exist in clear-text in system memory, while the protection of the Protected Execution would be removed as a result of the context loss of the CPU that is caused by the S3 power state. After a resume to normal operation is completed, the attacker would have access to the secret software located in clear-text in system memory.

If changing to power state S4 then the secret software would be be flushed to a hibernation file in clear-text. The attacker could procure access to the hibernation file and gain access to the secret software located in clear-text within the hibernation file.

To protect against this type of attack, the system (or to be more precise the Intel Trusted Execution Technology (TXT) platform) needs a method to mitigate the exposure of any system memory that could contain secrets to an attacker after a change in power state.

Should an ACPI sleep event occur while the Protected Execution is active, then the chipset responds with an LT-shutdown, which causes a complete platform reset. With the LT.HASSECRETS flag set, the protection mechanism that mitigates a Reset will be activated and SCLEAN will scrub the system memory to remove any secrets.

Method to achieve a ACPI enabled system in a secure way.

Should it be desirable to let the system be ACPI enabled while the Protected Execution is active, then the trusted software must intercept the ACPI sleep event and perform the

transition to a different power level in a secure way.

A secure power level transition could be accomplished like this:

1. ACPI sleep event is intercepted by the protected software.
2. Optionally any relevant context information is persisted.
3. The Protected Execution is terminated using the SEXIT operation.
   This operation removes the protected partition from system memory.
4. Change the power level to the specified ACPI sleep state (S3 or S4).
5. The system is now suspended until a resume from standby is signaled.
6. Resume to operational power state.
7. Execute the SENTER operation to resume Protected Execution.
8. Unseal the protected software.
9. Optionally any relevant context informations are retrieved from persisted storage.
10. Resume the disrupted execution of the protected software.

Performing the power level transition in this manor will ensure:

- Normal secure termination of Protected Execution before the standby occur. The result of this is that the standby has no security effects, as it does not occur while Protected Execution is active.
- The reestablishment of Protected Execution that occur after the system is resumed from standby is performed in a manor that exhibit the same security properties as an ordinary tamper-free execution of the protected software. The result is that the arguments that show the tamper-free execution is secure, will also show that resumed execution after standby is secure.

5.3 Future work

In this section I will mention some future work that could build upon the work of this report and extend the results achieved by this report.

Other Platforms
The protection method derived in this report is targeted against the PC platform. But the TPM specifications target a much wider range of platforms.
The platforms targeted by TCG in the TPM specifications are:

- Personal Digital Assistant (PDA) Devices
- Cellular and Mobile Devices
- Embedded Systems
- Storage Devices
- Infrastructure and Network Devices
- Virtualized Platforms

Future work could extend the derived protection method by analyzing the security properties of any of these other platforms, and describe how a tamper-free installation and execution of software could be achieved using the TPM specifications on these platforms.

Other CPU families than Intel
The protection method derived in this report is targeted against the Intel family CPU's. Other manufacturers of CPU are relevant on the Personal Computer (PC) platform. The two most prominent CPU manufacturers on this platform are Intel and AMD. Beside Intel that this report is targeted, the significant manufacturers of CPU families on the PC platform include:

- Advanced Micro Devices (AMD)
- Stanford University Network Microsystems (Sun)
- Motorola
- Transmeta

Future work could analyze the security properties of these CPU families, especially which method of Protected Execution the different CPU families support, as the ability of executing the protected software under a Protected Execution environment is a vital part of the proposed protection mechanism. The support and implementation methods for a Protected Execution scheme may vary considerable between the different CPU families in ways that affect the security properties of the protection services that is offered.

Build a Prototype
This report only derives a method for tamper-free installation and execution, and analyzes the security properties of the proposed methods. No software code is produced by this report. Future work could build a prototype by implementing the proposed method in code. This software could act as a proof of concept that illustrates how the proposed method could be implemented in practice. The creation of a prototype would enable the

validation of the proposed methods ability to withstand various attach types, not only in theory, but also in practice. The advantage of practically implemented attacks in contrast to the theoretical attacks, are that the actual implementation methods of the security concepts are tested. The theoretical analyzes can reveal weather it is possible to implement a solution that fulfill the required security properties, while a physical attack can reveal weather the actual implementation does fulfill the required security properties or not.
The following objects could be exposed to a physical attach to reveal the security properties of the actual implementation of the object:

- Trusted Platform Module (TPM) Device
- TPM Device Drivers
- TPM Device Driver Library
- TPM Software Stack (TSS)
- Protected Execution Software (aka SINIT)
- Prototype of the proposed method

The likely candidate of an operating system, to create the prototype under, would be Microsoft Windows, as software under this OS will receive the largest gain from a protection mechanism such as the proposed.

Types of hardware attack not covered

This report only address protection against software based attacks.
Examples of types of hardware attacks not covered:

- Attach probe and watch the Front Side Bus (FSB).
  The FSB provides the connection between the CPU and the Memory Controller Hub (MCH). The FSB is a fast bus with speeds of several MHz. Watching the FSB is possible assuming the attacker has the expertise and the equipment required. Reading the FSB will circumvent the protection provided by the Protected Execution.
- Attach probe and watch the Hublink.
  The Hublink provides the connection between the MCH and Input Output Controller Hub (ICH). The Hublink is a fast bus. Watching the Hublink is possible assuming the attacker has the expertise and the equipment required. The data on the Hublink is only the data in transit between the CPU and the TPM. Watching the Hublink is not worth the effort given the much less amount of work required to watch the LPC.
- Attach probe and watch the Low Pin Count Bus (LPC).
  The LPC provides the connection between the ICH and the TPM. The LPC is slow and very easy to watch. Data send under a Transport Session will be encrypted when traveling between the CPU and the TPM on the LPC.
- Freeze down and move system memory to an unsecured computer.
  System memory will hold the informations for several seconds after loss of power. By cooling down the system, it would be possible to extend the time until the information held in memory will degrade. Removing the system memory from a computer in a secure state at the time when the security sensible information is read

into memory, and transfer the memory to a unsecured computer, would give the attacker the possibility to access the security sensible informations that is held in memory. This attack would circumvent the protection provided by the Protected Execution.

Future work could include methods to mitigate or to make a hardware attack harder to execute successfully.

6 Conclusion

I will give a short summary on the process of creating this report.

Originally I had conceived that the creation of a sealed software package would take place only within the location of a server that was part of TCB. I imagined that the server could retrieve the informations required to create this package.
These informations would consist of :

- The SHA-1 hash of the trusted software.
- A non-migrateable key from the designated TPM.

Thus the secure software would never leave the Server in any other form than as a sealed package.
When I reached the part of the writing process where I analyzed the details of sealing and unsealing data within the TPM, I found that included in the sealed package was an encrypted value of tmpProof. This is a TPM secret that exists only in a TPM-shielded location, and never leaves the TPM in clear-text. The TPM uses this value to validate that a external data structure is actually created by this specific TPM.
As there is no way to retrieve the tpmProof value from a TPM, and a sealed package would not be valid without containing an encrypted copy of tpmProof, then it would not be possible to create a sealed package anywhere than inside the specific TPM.
To comply with the requirement of not to release the protected software outside the TCB in any other form than as a sealed package, I had to device another method of creating the sealed package. Using the TPM_Transport protocol I could expand the TCB to include the seal operation on the specified TPM, creating a secure communication tunnel, with the TPM as the end-point of the communication. This way, the clear-text form of the secure software will not leave the TCB during the process of creating the sealed package.

During the process of deriving possible attack schemes, I found a weakness in the way I used the Remote Attestation method.
The process of creating a sealed package consists of two steps:

- Perform Remote Attestation using DAA
- Create a Sealed software Package.

The attacker could allow the Remote Attestation to be performed with a trustworthy TPM, and then redirect the commands for creating a sealed software package to a Rogue TPM.
To eliminate this weakness, it was necessary for some binding to exist between the Remote Attestation and the Sealing commands. Using the TPM_Quote to sign a message with the AIK used in DAA, and including this quote in the Sealing command sequence, would create a binding between the attestation and the sealing operations. Only the designated TPM would know the private part of the AIK key and thus be able to perform the signing operation to give a valid response to the quote challenge.

The Hypothesis that this report is trying to confirm is:

Hypothesis: Using the TPM it will be possible to design a good mechanism that protects against tampering with the installation and execution of the protected software.

Verification of the hypothesis have been achieved by analyzing the possible attack methods against the proposed protection mechanism and argue or mathematically prove that the proposed method will protect against this attack.

It has been shown that it is computationally hard for an attacker to break the protection mechanism, so the goal of designing a good protection mechanism is considered successful.

The conclusion is that this report has shown that the hypothesis is correct, and the report has derived a method to solve the problem stated in the problem statement and has performed a theoretical description of the method.

References

[1] TPM Main-Part 2 TPM Structures_v1.2, TCG Published
[2] TPM Main-Part 1 Design Principles_v1.2, TCG Published
[3] TPM Main-Part 3 Commands_v1.2
[4] The Intel Safer Computing Initiative
[5] AMD64 Architecture Programmers Manual
[6] Martin Cochran, 2008, Notes on the Wang et al. 2^63 SHA-1 Differential Path
[7] Kiltz et.al, 2010, Instantiability of RSA-OAEP under Chosen-Plaintext Attack
[8] Wojtczuk & Rutkowska, November 2011, Attacking Intel TXT® via SINIT code execution hijacking
[9] TCG_PCClientTPMSpecification_v1-21
[10] Ernie Brickell et.al, Direct Anonymous Attestation
[11] Camenisch et.al, Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials

Glossary

Glossary

Attestation Identity Key (AIK)....22
AuthData....17
Authenticated Chipset Module (ACM)....39
Core Root of Trust for Measurement (CRTM)....15
Direct Anonymous Attestation (DAA)....22
Direct Memory Access (DMA)....36
Endorsement Key (EK)....6
Measured Virtual Machine Monitor (MVMM)....35
Non-migrateable Key....16
Nonce (Number Once)....33
Object-Specific Authorization Protocol (OSAP)....23
Platform Configuration Register (PCR)....35
Protected Execution Scheme....
SENTER....8
SKINIT....8
Root of Trust for Measurement (RTM)....15
SMI Transfer Module (STM)....37
Storage Root Key (SRK)....16
System Management Interrupts (SMI)....37
The protected software....34
The trusted software....34
Transport Session....25
Trusted Building Blocks (TBB)....15
Trusted Computing Group (TCG)....6
Trusted Graphics Translation Table (TGTT)....37
Trusted Platform Module (TPM)....6
Virtual Machine Monitor (VMM)....35

Appendix

A1 - TPM Commands and Structures

TPM_SEAL Command

Size	Type	Description
2	TPM_TAG	TPM_TAG_RQU_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_COMMAND_CODE	TPM_ORD_Seal
4	TPM_KEY_HANDLE	keyHandle
20	TPM_ENCAUTH	encrypted AuthData
4	UINT32	pcrInfoSize
?	TPM_PCR_INFO	pcrInfo
4	UINT32	inDataSize
?	BYTE[ ]	inData
4	TPM_AUTHHANDLE	OSAP Session authHandle
20	TPM_NONCE	nonceOdd
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	pubAuth

TPM_SEAL Result

Size	Type	Description
2	TPM_TAG	TPM_TAG_RSP_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_RESULT	returnCode
?	TPM_STORED_DATA	sealedData
4	TPM_AUTHHANDLE	OSAP Session authHandle
20	TPM_NONCE	nonceEven
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	resAuth

TPM_UNSEAL Command

Size	Type	Description

2	TPM_TAG	TPM_TAG_RQU_AUTH2_COMMAND
4	UINT32	paramSize
4	TPM_COMMAND_CODE	TPM_ORD_Unseal
4	TPM_KEY_HANDLE	keyHandle
?	TPM_STORED_DATA	inData
4	TPM_AUTHHANDLE	OSAP Session authHandle
20	TPM_NONCE	nonceOdd
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	parentAuth
4	TPM_AUTHHANDLE	dataAuthHandle
20	TPM_NONCE	dataNonceOdd
1	BOOL	continueDataSession
20	TPM_AUTHDATA	dataAuth

TPM_UNSEAL Result

Size	Type	Description
2	TPM_TAG	TPM_TAG_RSP_AUTH2_COMMAND
4	UINT32	paramSize
4	TPM_RESULT	returnCode
4	UINT32	secretSize
?	BYTE[ ]	secret
20	TPM_NONCE	nonceEven
1	BOOL	continueAuthSession
20	TPM_NONCE	datanonceEven
1	BOOL	continueDataSession
20	TPM_AUTHDATA	resAuth

TPM_EstablishTransport Command

Size	Type	Description
2	TPM_TAG	TPM_TAG_RQU_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_COMMAND_CODE	TPM_ORD_EstablishTransport
4	TPM_KEY_HANDLE	The handle to the key that encrypted the blob

?	TPM_TRANSPORT_PUBLIC	Information describing the transport session
4	UINT32	secretSize
?	BYTE[ ]	The encrypted secret area
4	TPM_AUTHHANDLE	authHandle
20	TPM_NONCE	nonceOdd
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	keyAuth

TPM_EstablishTransport Result

Size	Type	Description
2	TPM_TAG	TPM_TAG_RSP_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_RESULT	returnCode
4	TPM_TRANSHANDLE	The key handle for the Transport Session
4	TPM_MODIFIER_INDICATOR	The locality that called this command
32	TPM_CURRENT_TICKS	The current Tick count
20	TPM_NONCE	TransNonceEven
20	TPM_NONCE	nonceEven
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	resAuth

TPM_ExecuteTransport Command

Size	Type	Description
2	TPM_TAG	TPM_TAG_RQU_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_COMMAND_CODE	TPM_ORD_ExecuteTransport
4	UINT32	wrappedCmdSize
?	BYTE[ ]	wrappedCmd
4	TPM_TRANSHANDLE	The key handle for the Transport Session
20	TPM_NONCE	nonceOdd
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	transAuth

TPM_ExecuteTransport Result

Size	Type	Description
2	TPM_TAG	TPM_TAG_RSP_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_RESULT	returnCode
8	UINT64	The current ticks when the command executed
4	TPM_MODIFIER_INDICATOR	The locality that called this command
4	UINT32	wrappedRspSize
?	BYTE[ ]	The wrapped response
20	TPM_NONCE	nonceEven
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	resAuth

TPM_OwnerReadInternalPub Command

Size	Type	Description
2	TPM_TAG	TPM_TAG_RQU_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_COMMAND_CODE	TPM_ORD_OwnerReadInternalPub
4	TPM_KEY_HANDLE	Handle for either PUBEK or SRK
4	TPM_AUTHHANDLE	OSAP Session authHandle
20	TPM_NONCE	nonceOdd
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	pubAuth

TPM_OwnerReadInternalPub Result

Size	Type	Description
2	TPM_TAG	TPM_TAG_RSP_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_RESULT	returnCode
?	TPM_PUBKEY	The public portion of the requested key
4	TPM_AUTHHANDLE	OSAP Session authHandle
20	TPM_NONCE	nonceEven
1	BOOL	continueAuthSession

20	TPM_AUTHDATA	resAuth

TPM_GetPubKey Command

Size	Type	Description
2	TPM_TAG	TPM_TAG_RQU_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_COMMAND_CODE	TPM_ORD_GetPubKey
4	TPM_KEY_HANDLE	Key Handle
4	TPM_AUTHHANDLE	OSAP Session authHandle
20	TPM_NONCE	nonceOdd
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	pubAuth

TPM_GetPubKey Result

Size	Type	Description
2	TPM_TAG	TPM_TAG_RSP_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_RESULT	returnCode
?	TPM_PUBKEY	The public portion of the requested key
4	TPM_AUTHHANDLE	OSAP Session authHandle
20	TPM_NONCE	nonceEven
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	resAuth

TPM_Quote Command

Size	Type	Description
2	TPM_TAG	TPM_TAG_RQU_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_COMMAND_CODE	TPM_ORD_Quote
4	TPM_KEY_HANDLE	Key Handle to sign the PCR values
20	TPM_NONCE	ExternalData to prevent replay attacks
?	TPM_PCR_SELECTION	The PCRs that are to be reported
4	TPM_AUTHHANDLE	OSAP Session authHandle

20	TPM_NONCE	nonceOdd
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	pubAuth

TPM_Quote Result

Size	Type	Description
2	TPM_TAG	TPM_TAG_RSP_AUTH1_COMMAND
4	UINT32	paramSize
4	TPM_RESULT	returnCode
?	TPM_PCR_COMPOSITE	Structure containing the specified PCRs
4	UINT32	signSize
?	BYTE[ ]	The signed blob
20	TPM_NONCE	nonceEven
1	BOOL	continueAuthSession
20	TPM_AUTHDATA	resAuth

TPM_PCR_INFO Structure

Size	Type	Description
3	TPM_PCR_SELECTION	pcrSelection
20	TPM_COMPOSITE_HASH	digestAtRelease
20	TPM_COMPOSITE_HASH	digestAtCreation

TPM_PCR_SELECTION Structure

Size	Type	Description
2	UINT16	The size in bytes of the pcrSelect structure
1	BYTE	This SHALL be a bit map that indicates if a PCR is active or not

TPM_STORED_DATA Structure

Size	Type	Description
3	TPM_STRUCT_VER	This MUST be 1.1.0.0
4	UINT32	sealInfoSize
?	BYTE[ ]	sealInfo, structure of type TPM_PCR_INFO
4	UINT32	encDataSize

?	BYTE[ ]	encData

TPM_SEALED_DATA Structure

Size	Type	Description
1	TPM_PAYLOAD_TYPE	This MUST be TPM_PT_SEAL
20	TPM_SECRET	AuthData
20	TPM_SECRET	tmpProof
20	TPM_DIGEST	A digest of the TPM_STORED_DATA structure
4	UINT32	dataSize
?	BYTE[ ]	The data to be SEALED

TPM_STORE_ASYMKEY Structure

Size	Type	Description
1	TPM_PAYLOAD_TYPE	This MUST be TPM_PT_ASYM
20	TPM_SECRET	AuthData
20	TPM_SECRET	tmpProof
20	TPM_DIGEST	A digest of the TPM_KEY structure
4	UINT32	keySize
?	BYTE[ ]	The private key

A2 – Overview of tamper free method

A overview of the tamper free method is sketched in this rich diagram of the process: