ICS 35.030
CCS L 80
备案号：

中 华 人 民 共 和 国 密 码 行 业 标 准

GM/T XXXX—XXXX

证书透明规范

Certificate Transparency Specification

（草案）

在提交反馈意见时，请将您知道的相关专利连同支持性文件一并附上。

XXXX-XX-XX 发布　　XXXX-XX-XX 实施

国家密码管理局　发 布

目　　次

前　言

本标准按照GB/T 1.1—2020《标准化工作导则　第1部分:标准化文件的结构和起草规则》的规定起草。

请注意本标准的某些内容可能涉及专利。本标准的发布机构不承担识别专利的责任。

本标准由密码行业标准化技术委员会提出并归口。

本标准起草单位：零信技术（深圳）有限公司、证签技术（深圳）有限公司、北京航空航天大学网络空间安全学院、中国标准科技集团有限公司、重庆国家金融科技认证中心有限责任公司、华为技术有限公司、阿里云计算有限公司、三六零数字安全科技集团有限公司、贵州省电子认证科技有限公司、北京天威诚信电子商务服务有限公司、北京数字认证股份有限公司、上海市数字证书认证中心有限公司、广东省电子商务认证有限公司、新疆数字证书认证中心（有限公司）、河南省信息化集团有限公司、北京国富安电子商务安全认证有限公司、东方中讯数字证书认证有限公司、亚数信息科技（上海）有限公司。

本标准主要起草人：王高华、刘建伟、林诗杞、姬云鹏、李冰雨、丁元明、刘东华、秦逞、耿峰、耿东玉、龙勤、张天佳、张志磊、田勇、吕慧、张永强、戴业琪、陈树乐、李跃武、李亚飞、唐清文、李晓航、魏一才。

引　言

本标准参考RFC6962国际标准制定，但没有参考其升级标准RFC9162，主要是考虑到目前所有浏览器和CA机构都还在没有启用新版本，因为新版本改动非常大，我们制定标准的目的是尽快为商密SSL证书启用证书透明机制，待新的版本在国际算法SSL证书启用后再考虑依据RFC9162修订本标准。

证书透明规范

1 范围

本标准描述了一种商用密码数字证书的协议，用于公开记录签发的商密 SSL 证书的存在，其方式允许任何人审计 CA 机构的证书签发行为，并通过日志系统及时发现可疑证书的签发，CA 机构也可以自己审计证书日志。SSL 客户端可以拒绝信任未记录在日志中的证书，从而有效地迫使 CA 机构将所有已签发的证书提交到日志中透明公示。日志是实现本标准中定义的证书提交和查询协议的网络服务。

本标准适用于商用密码 SSL 证书的证书透明安全管理。

2 规范性引用文件

下列文件中的内容通过文中的规范性引用而构成本标准必不可少的条款。其中，注日期的引用文件，仅该日期对应的版本适用于本标准；不注日期的引用文件，其最新版本（包括所有的修改单）适用于本标准。

GB/T 16262.1-2006 信息技术 抽象语法记法一（ASN.1） 第一部份：基本记法规范

GB/T 20518-2018 信息安全技术 公钥基础设施 数字证书格式

GB/T 32905 信息安全技术 SM3密码杂凑算法

GB/T 32918（所有部份）信息安全技术 SM2椭圆曲线公钥密码算法

GM/Z 4001 密码术语

RFC 2119 Key words for use in RFCs to Indicate Requirement Levels

RFC 6962 Certificate Transparency

3 术语和定义

RFC 2119 和 RFC 6962 界定的以及下列术语、定义适用于本标准。

3.1

数据结构 data structure

数据结构是根据 RFC 5246 第 4 节中规定的约定定义的。

3.2

SSL 客户端 SSL Client

通常指使用 SSL 证书实现 https 加密的浏览器，也可以是移动 APP。

4 缩略语

下列缩略语适用于本标准。

ASN.1：抽象语法标记（Abstract Syntax Notation One）

CA：证书认证机构（Certificate Authority）

CRL：证书吊销列表（Certificate Revocation List）

TLS：传输层安全性协议（Transport Layer Security）
SSL：安全套接层（Secure Sockets Layer）
OID：对象标识符（Object Identifier）
RFC：请求意见稿（Request for Comments）
PKI：公钥基础设施（Public Key Infrastructure）

5 概述

证书透明旨在通过提供所有已签发的商用密码数字证书的可公开审计、仅追加的日志来减轻错误签发商用密码证书的问题。日志是可公开审计的，因此任何人都可以验证每个日志的正确性并监视何时向其中添加了新证书。日志本身并不能防止错误签发证书，但它们确保相关方(特别是那些在证书中绑定的名称)可以实时检测到此类错误签发。请注意，这是一种通用机制，但在本文档中，我们仅描述其用于由证书颁发机构 (CA) 签发的商密公共TLS/SSL服务器证书，简称为SSL证书。

每个日志由证书链组成，任何人都可以提交。公共CA应将其所有新签发的证书提交给一个或多个日志，包含签发此证书的完整证书链。为了避免日志被垃圾数据变成无用的垃圾，要求每个证书链都以日志信任的CA根证书为根。将证书链提交到日志时，将返回一个签名的时间戳，可用于向SSL客户端提供证书已提交透明备案的证据。因此，SSL客户端可以要求它们看到的所有证书都已在指定的日志系统备案。

那些担心错误签发证书的人可以监视日志，定期查询所有新条目，从而可以检查他们负责的域名是否已经签发了他们未知的SSL证书。他们如何处理这些信息，特别是当他们发现错误签发发生时，超出了本标准的范围，但从广义上讲，他们可以启用现有的CA业务机制来处理错误签发的证书。当然，任何人都可以监视日志，如果他们认为证书签发不正确，可以采取他们认为合适的措施。国家密码管理部门可以利用日志数据实时掌握各个CA的商密SSL证书的签发情况，包括哪个CA机构为哪个域名何时签发了商密SSL证书，并实时监督各个CA的商密SSL证书签发是否存在违规行为。

同样，那些从某个日志中看到签名时间戳的人可以要求从该日志中获得透明备案证明。如果日志无法提供这一点(或者实际上，如果监视方无法从该日志副本中找到相应的证书)，那就是日志系统操作不当的证据，以此监督日志本身是否存在违规行为。

每个日志的append-only(只能追加)属性在技术上是使用默克尔树实现的，它可以用来表明日志的任何某个版本都是任何某个先前版本的超集。同样，默克尔树避免了盲目信任日志的需求：如果日志试图向不同的人展示不同的东西，这可以通过比较树根和一致性证明来有效地检测到。类似地，任何日志的其他不当行为(例如，为它们随后不记录的证书签发签名时间戳)可以被有效地检测并暴露在公众面前。这有效地制约了日志系统本身的安全可信，只有这样才能放心地用于监视CA机构的SSL证书签发行为是否可信。

6 密码组件

6.1 哈希算法和默克尔(Merkle)哈希树

日志使用二进制默克尔哈希树进行高效审计。哈希算法是 SM3[GB/T 32905]。默克尔树哈希的输入是一个数据条目列表；这些条目将被哈希以形成默克尔哈希树的叶子。输出是一个 32 字节的默克尔树哈希。给定 n 个输入的有序列表，D[n] = {d(0), d(1), ..., d(n-1)}，默克尔树哈希 (MTH)定义如下：

空列表的哈希是空字符串的哈希：

MTH({}) = SM3()。

具有一个条目的列表的哈希(也称为叶哈希)是：

MTH({d(0)}) = SM3(0x00 || d(0))。

对于 n>1，令 k 为小于 n 的两个的最大幂(即，k<n <= 2k)。然后将 n 元素列表 D[n]的默克尔树哈希递归定义为：

MTH(D[n]) = SM3(0x01 || MTH(D[0:k]) || MTH(D[k:n])),

这里||是串联，D[k1:k2]表示长度为(k2 - k1)的列表{d(k1), d(k1+1),..., d(k2-1)}。(请注意，叶子和节点的哈希计算不同，这两种情况需要分开处理以提供抗二次原像性（second preimage resistance)。

请注意：我们不要求输入列表的长度是 2 的幂。由此产生的默克尔树可能不平衡；然而，它的形状是由叶子的数量唯一决定的。

6.1.1 默克尔审计路径

默克尔哈希树中叶子的默克尔审计路径是计算该树的默克尔树哈希所需的默克尔树中附加节点的最短列表。树中的每个节点要么是叶节点，要么是从其正下方(即朝向叶)的两个节点计算而来。在树的每一步(向根)，审计路径中的一个节点与到目前为止计算的节点相结合。换句话说，审计路径由计算从叶子到树根的节点所需的缺失节点列表组成。如果从审计路径计算出的根与真正的根相匹配，则审计路径证明叶子存在于树中。

给定树的 n 个输入的有序列表 D[n] = {d(0), ..., d(n-1)}，默克尔审计路径 PATH(m, D[n])对于 (第 m+1)个输入 d(m)，0 <= m < n，定义如下：

具有单元素输入列表 D[1] = {d(0)}的树中单个叶子的路径为空：

PATH(0, {d(0)}) = {}

对于 n > 1，令 k 为小于 n 的两个中的最大幂。n > m 元素列表中第(m+1)个元素 d(m)的路径递归定义为：

PATH(m, D[n]) = PATH(m, D[0:k]) : MTH(D[k:n]) for m < k;和

PATH(m, D[n]) = PATH(m - k, D[k:n]) : MTH(D[0:k]) for m >= k,

其中：是列表的串联，D[k1:k2]表示长度(k2 - k1)列表{d(k1), d(k1+1),..., d(k2-1)}和以前一样。

6.1.2 默克尔一致性证明

默克尔一致性证明证明了树的只可追加属性。 默克尔树哈希 MTH(D[n])的默克尔一致性证明和先前公布的前 m 个叶子的哈希 MTH(D[0:m])，m <= n，是 默克尔树中的节点列表需要验证前 m 个输入 D[0:m] 在两棵树中是否相等。因此，一致性证明必须包含一组足以验证 MTH(D[n])的中间节点（即对输入的承诺)，这样(的一个子集)相同节点可用于验证 MTH(D[0 :m]).我们定义了一个算法来输出(唯一

的)最小一致性证明。

给定树的 n 个输入的有序列表，D[n] = {d(0), ..., d(n-1)}，Merkle 一致性证明 PROOF(m, D[n]) 对于先前的默克尔树哈希 MTH(D[0:m]), 0 < m < n, 定义为：

PROOF(m, D[n]) = SUBPROOF(m, D[n], true)

如果 m 是最初请求 PROOF 的值，则 m = n 的子证明为空(意味着子树的默克尔树哈希 MTH(D[0:m]) 已知)：

SUBPROOF(m, D[m], true) = {}

m = n 的子证明是 默克尔树哈希提交输入 D[0:m]；否则：

SUBPROOF(m, D[m], false) = {MTH(D[m])}

对于 m < n，令 k 为小于 n 的两个的最大幂。然后递归定义子证明。

如果 m <= k，则右子树条目 D[k:n]仅存在于当前树中。我们证明左子树条目 D[0:k]是一致的，并向 D[k:n]添加承诺：

SUBPROOF(m, D[n], b) = SUBPROOF(m, D[0:k], b) : MTH(D[k:n])

如果 m > k，左子树条目 D[0:k]在两棵树中是相同的。我们证明右子树条目 D[k:n]是一致的，并向 D[0:k]添加承诺。

SUBPROOF(m, D[n], b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k])

这里，: 是列表的串联，D[k1:k2]表示长度 (k2 - k1)列表{d(k1), d(k1+1),..., d(k2-1)}为前。

结果证明中的节点数以 ceil(log2(n)) + 1 为界。

6.1.3 举例

有 7 个叶子的二叉默克尔树（见图 1）：

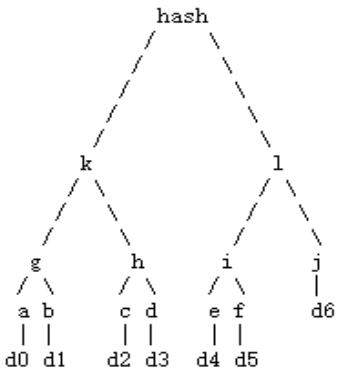

图 1 7 个叶子的二叉默克尔树

d0 的审计路径是[b, h, l]。
d3 的审计路径是[c, g, l]。
d4 的审计路径是[f, j, k]。
d6 的审计路径是[i, k]。

同一棵树，分四个步骤逐步构建（见图 2）：

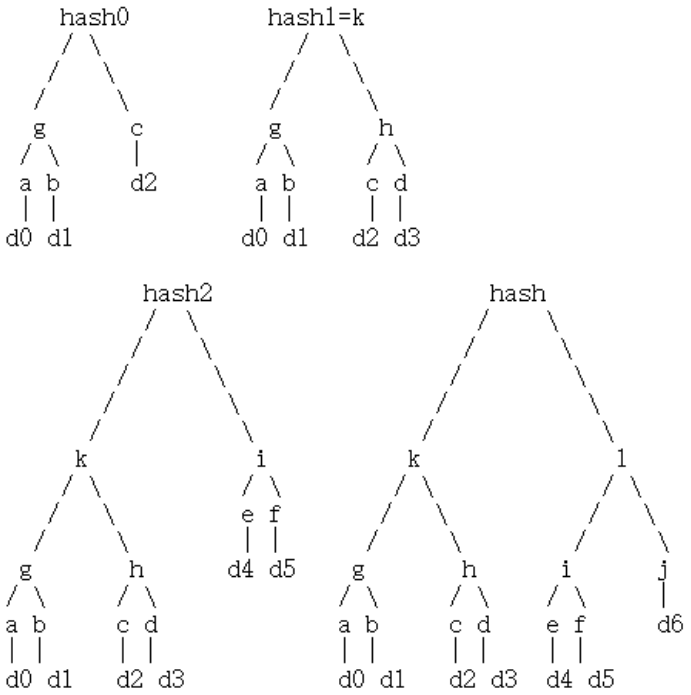

图 2　默克尔树构建示意图

hash0 与 hash 的一致性证明为 PROOF(3, D[7]) = [c, d, g, l]。 c、g 用于验证 hash0，d、l 用于额外证明 hash 与 hash0 一致。

hash1 和 hash 的一致性证明为 PROOF(4, D[7]) = [l]。hash 可以使用 hash1=k 和 l 来验证。

hash2 和 hash 的一致性证明是 PROOF(6, D[7]) = [i, j, k]。k，i 用于验证 hash2，j 用于额外证明 hash 与 hash2 一致。

6.1.4　签名

各种数据结构必须签名。日志必须使用 GB/T 32918 的 SM2 椭圆曲线签名。

7　日志格式及操作

7.1　概述

任何人都可以将证书提交到日志以供公开审计；然而，由于除非已记录，否则 SSL 客户端不会信任此证书，因此证书所有者或其 CA 必须提交它们。日志是此类证书的一个单一的、不断增长的、只能追加的默克尔树。

当一个有效的证书或预签证书被提交到日志时，日志必须立即返回一个签名时间戳(SCT)。SCT 是日志承诺在称为最大合并延迟（MMD）的固定时间内将证书合并到默克尔树中的证明。如果日志之前已经记录证书，它可能会返回与之前返回的相同的 SCT。Web 服务器必须将来自一个或多个日志的 SCT 连同证书一起提供给 SSL 客户端。SSL 客户端必须拒绝没有有效 SCT 的用户证书。

每个日志定期将其所有新条目附加到默克尔树中，并对树的根进行签名。因此，审计方可以验证已包含 SCT 的每张证书确实出现在日志中。日志必须在 SCT 发布后的最大合并延迟期间内将证书合并到其默克尔树中。

日志运营方不得对从日志中检索或共享数据强加任何条件。

7.2 日志条目

任何人都可以向日志提交证书。为了能够将每个记录的证书归于其证书签发者，日志应发布可接受的根证书列表(该列表可能是浏览器信任的根证书列表)。每个提交的证书必须附有完整的证书链，直至根证书列表中的根证书。

证书签发机构必须在签发之前将证书提交到日志系统。为此，CA 提交一个预签证书(Precertificate)，日志可以使用该预签证书创建一个条目，该条目将对已签发的证书有效。预签证书是通过向用户证书添加一个特殊的关键毒丸扩展字段(OID 1.2.156.10197.2.4.3(假设值，待实际分配，以下同)，其 extnValue OCTET STRING 包含 ASN.1 NULL 数据(0x05 0x00))，这个扩展字段是为了确保预签证书不能被标准的 X.509 v3 客户端验证，并采用以下两种方式对 TBSCertificate[GB/T 20518-2018]签名：

a) 特殊用途(CA:true, Extended Key Usage: Certificate Transparency, OID 1.2.156.10197.2.4.4)(待分配)预签证书签名证书。这张预签证书签名证书必须由（根证书或中级根证书）CA 证书直接认证，该证书最终将用于签名用户证书的 TBSCertificate，以产生最终用户证书(请注意,日志可能会放宽标准验证规则以允许这样做,只要签发的证书将有效)；
b) 用于 CA 证书签发用户证书。

如上所述，预签证书提交必须附有预签证书签名证书（如果使用）以及验证证书链到接受的根证书所需的所有其他证书。TBSCertificate 上的签名表示证书签发机构签发证书的意图。此意图被视为具有约束力(即：错误签发预签证书被视为等同于错误签发最终用户证书)。每个日志都会验证预签证书的签名链，并在相应的 TBSCertificate 上签发已签证书时间戳(SCT)。

日志必须使用提交者提供的中级根 CA 证书链，验证提交的用户证书或 预签证书是否具有通向受信任根 CA 证书的有效签名链。日志可以接受已过期、尚未生效、已被撤销或根据 X.509 验证规则不完全有效的证书，以适应 CA 系统的各种证书签发情况。但是，日志必须拒绝发布没有有效证书链到已知根 CA 的证书。如果接受证书并签发 SCT，接受日志必须存储用于验证的整个证书链，包括用户证书或预签证书本身，包括用于验证证书链的顶级根证书(即使它在提交中被省略)，并且必须根据要求提供此证书链以供审计。需要此证书链来防止 CA 通过记录部分或空链来逃避责备。（注意：这有效地排除了自签名和基于 DANE 的证书，直到找到某种机制来控制减少验证这些证书的多余数据。）

日志中的每张证书条目必须包含以下组件：

```
enum { x509_entry(0), precert_entry(1), (65535) } LogEntryType;

struct {
          LogEntryType entry_type;
          select (entry_type) {
              case x509_entry: X509ChainEntry;
```

```
            case precert_entry: PrecertChainEntry;
        } entry;
    } LogEntry;

    opaque ASN.1Cert<1..2^24-1>;

    struct {
        ASN.1Cert leaf_certificate;
        ASN.1Cert certificate_chain<0..2^24-1>;
    } X509ChainEntry;

    struct {
        ASN.1Cert pre_certificate;
        ASN.1Cert precertificate_chain<0..2^24-1>;
    } PrecertChainEntry;
```

日志可以限制他们接受的链的长度。

“entry_type”是这个条目的类型。此协议版本的未来修订可能会添加新的 LogEntryType 值。第 7 节解释了客户端应该如何处理未知的条目类型。

“leaf_certificate”是提交审计的最终用户证书。

“certificate_chain”是验证最终用户证书所需的附加证书链。第一张证书必须是最终用户证书，每个后续证书必须直接证明它前面的证书，最后一张证书必须是日志接受的顶级根证书。

“pre_certificate”是提交审计的预签证书。

“precertificate_chain”是验证预签证书提交所需的附加证书链。第一张证书必须是一个有效的预签证书签名证书并且必须能证明预签证书，每个后续证书必须直接证明它前面的证书，最后一张证书必须是日志接受的顶级根证书。

7.3 已签证书时间戳(SCT)的结构

```
enum { certificate_timestamp(0), tree_hash(1), (255) }
  SignatureType;

enum { v1(0), (255) }
  Version;

struct {
    opaque key_id[32];
} LogID;

opaque TBSCertificate<1..2^24-1>;

struct {
  opaque issuer_key_hash[32];
  TBSCertificate tbs_certificate;
```

```
        } PreCert;

        opaque CtExtensions<0..2^16-1>;
```

“key_id”是日志公钥的SM3哈希值，通过表示为SubjectPublicKeyInfo的密钥的DER编码计算得出。

“issuer_key_hash”是证书签发者公钥的SM3哈希，根据表示为SubjectPublicKeyInfo的密钥的DER编码计算得出。这是将签发者绑定到最终用户证书所必需的。

“tbs_certificate”是预签证书的DER编码TBSCertificate组件--也就是说，没有签名和毒丸扩展项。如果预签证书未使用将签发最终用户证书的CA证书签名，则TBSCertificate 也将其签发者更改为将签发最终用户证书CA证书。请注意，也可以通过从最终用户证书中提取 TBSCertificate 并删除SCT扩展来重建此TBSCertificate。另请注意，由于TBSCertificate包含必须匹配预签证书签名算法和最终用户证书签名算法的AlgorithmIdentifier，因此必须使用相同的算法和参数对它们进行签名。如果预签证书是使用预签证书签名证书签发的，并且TBSCertificate中存在授权密钥标识符扩展，则相应的扩展也必须存在于预签证书签名证书中--在这种情况下TBSCertificate的授权密钥标识符也更改为匹配的最终签发CA。

```
struct {
           Version sct_version;
           LogID id;
           uint64 timestamp;
           CtExtensions extensions;
           digitally-signed struct {
               Version sct_version;
               SignatureType signature_type = certificate_timestamp;
               uint64 timestamp;
               LogEntryType entry_type;
               select(entry_type) {
                   case x509_entry: ASN.1Cert;
                   case precert_entry: PreCert;
               } signed_entry;
              CtExtensions extensions;
           };
       } SignedCertificateTimestamp;
```

数字签名元素的编码在[RFC5246]中定义。

“sct_version”是SCT遵循的协议版本。这个版本是v1。

“timestamp”是当前的NTP时间[RFC5905]，从纪元(1970年1月1日，00:00)开始计量，忽略闰秒，以毫秒为单位。

“entry_type” 可以从呈现SCT的上下文中隐含。

“signed_entry” 是“leaf_certificate”(在 X509ChainEntry 的情况下)或者是 PreCert(在PrecertChainEntry的情况下)，如上所述。

“extensions”是此协议版本(v1)的未来扩展。目前，没有指定扩展名。

7.4 在 SSL 握手中包含 SCT

在 SSL 握手中必须包含至少来自一个日志的最终用户证书相对应的 SCT 数据，方法是使用如下所述的 X509v3 证书扩展项，或使用 SSL 扩展([RFC5246]的第 7.4.1.4 节）键入“signed_certificate_timestamp”，或使用在线证书状态协议(OCSP)装订(也称为“证书状态请求”SSL 扩展；参见[RFC6066])，其中响应包括 OID 为 1.2.156.10197.2.4.5 的 OCSP 扩展(参见[RFC2560])和正文：

```
SignedCertificateTimestampList ::= OCTET STRING
```

必须至少包含一个 SCT，可以包括多个 SCT。

类似地，证书签发机构可以向多个日志提交预签证书，并且所有获得的 SCT 可以直接嵌入到最终用户证书中，方法是将 SignedCertificateTimestampList 结构编码为 ASN.1 OCTET STRING，并将结果数据作为 TBSCertificate 插入 X.509v3 证书扩展(OID 1.2.156.10197.2.4.2)。收到证书后，客户端可以重建原始的 TBSCertificate 来验证 SCT 签名。

OCSP 扩展或 X509v3 证书扩展中嵌入的 ASN.1 OCTET STRING 的内容如下：

```
opaque SerializedSCT<1..2^16-1>;

struct {
    SerializedSCT sct_list <1..2^16-1>;
} SignedCertificateTimestampList;
```

这里，“SerializedSCT”是一个包含序列化 SSL 结构的不透明字节字符串。这种编码确保 SSL 客户端可以单独解码每个 SCT(即，如果有版本升级，过时的客户端仍然可以解析旧的 SCT，同时跳过它们不理解其版本的新 SCT)。

同样，SCT 可以嵌入到 SSL 扩展中。详情见下文。

SSL 客户端必须实现所有三种机制。SSL 服务端必须至少实现三种机制中的一种。请注意，现有的 SSL 服务端通常可以不加修改地使用证书扩展机制。

SSL 服务端应该同时发送多个 SCT，以防一个或多个日志不被客户端接受(例如，如果日志因不当行为而被删除或有密钥泄露)。

7.4.1 SSL 扩展

SCT 可以在 SSL 握手期间使用类型为“signed_certificate_timestamp”的 SSL 扩展方式发送。

支持扩展的客户端应该发送具有适当类型和空“extension_data”的 ClientHello 扩展。

服务端必须只向已在 ClientHello 中指示支持扩展的客户端发送 SCT，在这种情况下，通过将“extension_data”设置为“SignedCertificateTimestampList”来发送 SCT。

会话恢复使用原始会话信息：客户端应该在 ClientHello 中包含扩展类型，但如果会话恢复，则服务端不应处理它或在 ServerHello 中包含扩展。

7.5 默克尔树

默克尔树哈希的哈希算法是 SM3。

默克尔树输入的结构：

```
enum { timestamped_entry(0), (255) }
  MerkleLeafType;

struct {
    uint64 timestamp;
    LogEntryType entry_type;
    select(entry_type) {
        case x509_entry: ASN.1Cert;
        case precert_entry: PreCert;
    } signed_entry;
    CtExtensions extensions;
} TimestampedEntry;

struct {
    Version version;
    MerkleLeafType leaf_type;
    select (leaf_type) {
        case timestamped_entry: TimestampedEntry;
    }
} MerkleTreeLeaf;
```

这里，“version”是 MerkleTreeLeaf 对应的协议版本。这个版本是 v1。

“leaf_type”是叶子输入的类型。目前只定义了“timestamped_entry”(对应一个 SCT)。此协议版本的未来修订可能会添加新的 MerkleLeafType 类型。第 7 节解释了客户端应该如何处理未知的叶子类型。

“timestamp”是为此证书签发的对应 SCT 的时间戳。

“signed_entry”是相应 SCT 的“signed_entry”。

“extensions”是相应 SCT 的“扩展”。

默克尔树的叶子是相应“MerkleTreeLeaf”结构的叶子哈希。

7.6 已签名树头(STH)

每次日志向树添加新条目时，日志应该对相应的树哈希和树信息进行签名(参见第 7.4 节中相应的签名树头客户端消息)。该数据的签名结构如下：

```
digitally-signed struct {
        Version version;
        SignatureType signature_type = tree_hash;
```

```
        uint64 timestamp;
        uint64 tree_size;
        opaque sm3_root_hash[32];
    } TreeHeadSignature;
```

“version”是 TreeHeadSignature 所遵循的协议版本。这个版本是 v1。

“timestamp”是当前时间。时间戳必须至少与树中最新的 SCT 时间戳一样。每个后续时间戳必须比前一个更新的时间戳还要新。

“tree_size”等于新树中的条目数。

“sm3_root_hash”是默克尔哈希树的根。

每个日志必须按需生成不早于最大合并延迟(MMD)的签名树头。如果它在 MMD 期间没有收到新的证书提交，日志应使用新的时间戳签名相同的默克尔树哈希。

8 日志的客户端消息

8.1 概述

消息以 HTTPS GET 或 POST 请求发送。POST 的参数和所有响应都被编码为 JavaScript 对象表示法(JSON)对象[RFC4627]。GET 的参数被编码为与顺序无关的键/值 URL 参数，使用“HTML 4.01 规范”[HTML401]中描述的“application/x-www-form-urlencoded”格式。二进制数据采用 base64 编码[RFC4648]，如各个消息中指定的那样。

请注意，JSON 对象和 URL 参数可能包含此处未指定的字段。这些额外的字段应该被忽略。

<log server>前缀可以包含路径以及日志服务器名称和端口。

一般来说，在需要的地方，“version”是 v1，“id”是查询的日志服务器的 log id。

任何错误都将作为 HTTP 4xx 或 5xx 响应返回，并带有人类可读的错误消息。

8.2 将证书链添加到日志

POST https://<log server>/ct/v1/add-chain

输入：

chain：一组 base64 编码的证书。第一个元素是最终用户证书；第二个是中级根证书，依此类推到最后一个，即顶级根证书或链接到已预置的根证书。

输出：

sct_version：SignedCertificateTimestamp 结构的版本，十进制。目前为 v1。

id：日志 ID，base64 编码。由于请求 SCT 以包含在 SSL 握手中的日志客户端不需要对其进行验证，因此我们不假设他们知道日志的 ID。

timestamp：SCT 时间戳，十进制。

extensions：一种不透明的类型，用于将来的扩展。很可能并非所有参与者都需要了解该领域的数据。日志应将其设置为空字符串。客户端应解码 base64 编码的数据并将其包含在 SCT 中。

signature：SCT 签名，base64 编码。如果“sct_version”不是 v1，则 v1 客户端可能无法验证签名。它绝不能将此解释为错误。(注意：日志客户端不需要能够验证此结构；只有 SSL 客户端需要。如果我们将结构作为二进制 blob 提供，那么我们可以完全更改它而无需升级到 v1 客户端。)

8.3 添加预签证书链(PreCertChain)到日志

POST https://<log server>/ct/v1/add-pre-chain

输入：

chain：一组 base64 编码的预签证书和根证书。第一个元素是预签用户证书；第二个链接到第一个，依此类推到最后一个，即顶级根证书或链接到已预置的根证书。

输出：

与第 7.2 节中的相同。

8.4 获取最新 STH

GET https://<log server>/ct/v1/get-sth

没有输入。

输出：

tree_size：树的大小，以条目为单位，以十进制表示。

timestamp：时间戳，十进制。

sm3_root_hash：默克尔哈希树的根，base64。

tree_head_signature：上述数据的 TreeHeadSignature。

8.5 获取两个 STH 之间的默克尔一致性证明

GET https://<log server>/ct/v1/get-sth-consistency

输入：

first：第一棵树的 tree_size，十进制。

second：第二棵树的 tree_size，十进制。

两种树的大小都必须来自现有的 v1 STH。

输出：

consistency：默克尔树节点数组，base64 编码。

请注意，此数据不需要签名，因为它用于验证已签名的 STH。

8.6 通过叶哈希从日志中获取默克尔审计证明

GET https://<log server>/ct/v1/get-proof-by-hash

输入：

hash：base64 编码的 v1 叶哈希。

tree_size：证明所基于的树的 tree_size，以十进制表示。

“hash”必须按照第 6.5 节中的定义进行计算。“tree_size”必须指定现有的 v1 STH。

输出：

leaf_index：“hash”参数对应的是从 0 开始的索引。

audit_path：一组 base64 编码的默克尔树节点，证明包含所选证书。

8.7 从日志中获取条目

GET https://<log server>/ct/v1/get-entries

输入：

start：要检索的第一个条目的从 0 开始的索引，以十进制表示。

end：要检索的最后一个条目的从 0 开始的索引，以十进制表示。

输出：

entries：对象数组，每个对象由下面的两个数据组成：

leaf_input：base64 编码的 MerkleTreeLeaf 结构。

extra_data：与日志条目相关的 base64 编码的无符号数据。在 X509ChainEntry 的情况下，

这是“certificate_chain”。在 PrecertChainEntry 的情况下，这是整个“PrecertChainEntry”。

请注意，此消息未签名--可以通过构造与检索到的 STH 对应的默克尔树哈希来验证检索到的数据。所有叶子都必须是 v1。但是，兼容的 v1 客户端不得将无法识别的 MerkleLeafType 或 LogEntryType 值解释为错误。这意味着它可能无法解析某些条目，但请注意，每个客户端都可以检查它能识别的条目，并通过将无法识别的叶子视为树的不透明输入来验证数据的完整性。

“start”和“end”参数应该在 0 <= x < “tree_size”范围内，如第 7.4 节中“get-sth”返回的那样。

日志可以通过返回仅涵盖指定范围内的有效条目的部分响应来符合 0 <= "start" < "tree_size" 和 "end" >= "tree_size"的要求。请注意，以下限制也可能适用：

日志可以限制每个“get-entries”请求可以检索的条目数。如果客户端请求的条目数超过允许的条目数，则日志应返回允许的最大条目数。这些条目应按顺序从“start”指定的条目开始。

8.8 获取已预置信任的根证书

GET https://<log server>/ct/v1/get-roots

没有输入。

输出：

certificates：日志可接受的一组 base64 编码的根证书。

8.9 从日志中获取条目+默克尔审计证明

GET https://<log server>/ct/v1/get-entry-and-proof

输入：

leaf_index：所需条目的索引。

tree_size：需要证明的树的 tree_size。

树的大小必须指定现有的 STH。

输出：

leaf_input：base64 编码的 MerkleTreeLeaf 结构。

extra_data：base64 编码的无符号数据，同 7.7 节。

audit_path：一组 base64 编码的默克尔树节点，证明包含所选证书。

这个 API 可能只对调试有用。

9 日志用户

9.1 概述

日志用户可能执行各种不同的功能。我们在这里描述了一些典型的用户以及他们是如何运作的。任何不一致都可以用作日志行为不当的证据，并且数据结构上的签名可以防止日志否认该不当行为。

所有日志用户都应该互相交流，至少交换 STH；这就是确保他们都具有一致视图所需要的全部信息。交流的确切机制将在单独的文件中描述，但预计会有多种。

9.2 日志提交者

日志提交者如上所述向日志提交证书或预签证书。他们可能会继续使用返回的 SCT 来构建证书或直接在 SSL 握手中使用它。日志提交者通常为 CA 机构，也可以是其他方。

9.3 SSL 客户端

SSL 客户端通常是指支持商密算法 SSL 证书的浏览器或移动 APP，它们会与服务器 SSL 证书一起或在服务器 SSL 证书中接收 SCT。除 SSL 证书及其证书链的正常验证外，它们还应通过计算从 SCT 数据输入的签名以及证书并使用相应日志的公钥验证签名来验证 SCT。请注意，本文档并未描述客户端如何获取日志的公钥。

SSL 客户端必须拒绝时间戳的时间是将来的时间的 SCT 和不信任这样的 SSL 证书。

9.4 监视方

监视方监视日志并检查它们是否正确运行。他们还关注感兴趣的证书。

监视方至少需要检查它监视的每个日志中的每个新条目。它可能还想保留整个日志的副本。为此，应针对每个日志执行以下步骤：

a) 获取当前 STH(第 7.4 节)；
b) 验证 STH 签名；
c) 获取树中与 STH 对应的所有条目(第 7.7 节)；
d) 确认从获取的条目生成的树产生与 STH 中相同的哈希；
e) 获取当前 STH(第 7.4 节)，重复直到 STH 改变；
f) 验证 STH 签名；
g) 获取树中与 STH 对应的所有新条目(第 7.7 节)。如果长时间无法获取新条目，则应该被视为日志的不当行为之一；
h) 或者：
    1) 验证更新后的所有条目列表是否生成了一棵与新 STH 具有相同哈希的树；或者，如果它不保留所有日志条目：
    2) 获取新 STH 与先前 STH 的一致性证明(第 7.5 节)；
    3) 验证一致性证明；
    4) 验证新条目是否生成一致性证明中的相应元素。
i) 转到步骤 e。

9.5 审计方

审计方将有关日志的部分信息作为输入，并验证此信息与他们拥有的其他部分信息是否一致。审计方可能是 SSL 客户端的组成部分；它可能是一个独立的服务；或者它可能是监视方的次要功能。

来自同一日志的任何一对 STH 都可以通过请求一致性证明来验证(第 7.5 节)。

通过请求默克尔审计证明，带有 SCT 的证书通过与 SCT 时间戳+最大合并延迟之后的任何 STH 进行验证(第 7.6 节)。

当然，审计方可以不时自行获取 STH(第 7.4 节)。

10 IANA 相关事项

IANA 已为 SCT SSL 扩展分配了 RFC 5246 ExtensionType 值（18），扩展名为“signed_certificate_timestamp”。

11 安全注意事项

11.1 概述

通过 CA、日志和 Web 服务器执行此处描述的操作，SSL 客户端可以使用日志和签名时间戳来降低它们接受错误签发的证书的可能性。如果 Web 服务器为证书提供有效的签名时间戳，则客户端知道该证书已在日志中发布。由此，客户端用户就有时间在发现错误签发时根据证书主题信息采取一些行动，例如要求 CA 吊销错误签发的证书。证书透明并不能保证证书不会被错误签发和不会被部署使用，因为证书的主体可能没有检查日志或者 CA 可能拒绝吊销证书，所以 SSL 客户端必须给出明确的证书未透明备案的安全提示。

此外，要求 SSL 客户端不接受未透明记录的证书，这样 CA 机构就更有动力将证书提交到日志中，从而提高 SSL 证书生态的整体透明度。

11.2 错误签发的证书

未公开记录且因此没有有效 SCT 的错误签发的证书将被 SSL 客户端拒绝。假设日志运行正常，错误签发的具有来自日志的 SCT 的证书将在最大合并延迟时间内出现在该公共日志中。因此，可以使用错误签发的证书而不被发现的最长时间是最大合并延迟时间。

11.3 错误检查

日志本身不会检测错误签发的证书；相反，他们依靠相关方(例如域名所有者，密码管理机构)来监视他们并在检测到错误时采取纠正措施。

11.4 行为不端的日志

日志可能以两种方式表现不当行为：(1)未能在最大合并延迟时间内将证书与 SCT 合并到默克尔树中，以及(2)通过在不同时间和/或向不同方呈现默克尔树的两个不同的、相互冲突的视图来违反其仅追加属性。这两种形式的违规行为都将被迅速公开地发现。

日志客户端通过检索每个观察到的 SCT 的默克尔审计证明，就可以检测到违反最大合并延迟的规定的情况。这些检查可以是异步的，每张证书只需执行一次。为了保护用户隐私，这些检查不需要向日志显示确切的证书。用户可以改为向受信任的审计方请求审计证明(因为任何人都可以从日志中计算审计证明)，或者为 SCT 时间戳周围的一批证书请求默克尔证明。

违反仅附加属性的行为会被公众检测到，即每个人都在审计日志中比较他们最新的签名树头的版本。一旦从同一日志中检测到两个冲突的 STH，这是该日志有不当行为的确切密码技术证明。

12 效率考虑

默克尔树设计的目的是保持低通信开销。

审计日志的完整性不需要第三方维护每个完整日志的副本。可以在新条目可用时更新签名树头，而无需重新计算整棵树。第三方审计只需针对日志的现有 STH 获取默克尔一致性证明，即可有效验证其默克尔树更新的仅追加属性，而无需审计整棵树。

参 考 文 献

[1] [CrosbyWallach] Crosby, S.和D. Wallach，“防篡改日志记录的高效数据结构”，第18届USENIX安全研讨会论文集，蒙特利尔，2009年8月，< http://static.usenix.org/event/sec09 / tech/full_papers/crosby.pdf>。

[2] [DSS] 美国国家标准技术研究院，“数字签名标准(DSS)”，FIPS 186-3，2009年6月，<http://csrc.nist.gov/publications/fips/ fips186-3/fips_186-3.pdf>。

[3] [FIPS.180-4] 美国国家标准技术研究院，“安全哈希标准”，FIPS PUB 180-4，2012年3月，<http://csrc.nist.gov/publications/fips/fips180-4/ fips-180-4.pdf>。

[4] [HTML401] Raggett, D.、Le Hors, A.和I. Jacobs，“HTML 4.01规范”，万维网联盟推荐REC-html401-19991224，1999年12月，<http://www.w3.org/ TR/1999/REC-html401-19991224 >。

[5] [RFC2560] Myers, M.、Ankney, R.、Malpani, A.、Galperin, S.和C. Adams，“X.509互联网公钥基础设施在线证书状态协议-OCSP”， RFC 2560，1999年6月。

[6] [RFC3447] Jonsson, J.和B. Kaliski，“公钥加密标准 (PKCS) #1：RSA 加密规范版本2.1”， RFC 3447，2003年2月。

[7] [RFC4627] Crockford, D.，“用于JavaScript对象表示法(JSON)的application/json媒体类型”，RFC 4627，2006年7月。

[8] [RFC4648] Josefsson, S.，“Base16、Base32和Base64数据编码”，RFC 4648，2006年10月。

[9] [RFC5246] Dierks, T.和E. Rescorla，“传输层安全(TLS)协议版本1.2”，RFC 5246，2008年8月。

[10] [RFC5280] Cooper, D.、Santesson, S.、Farrell, S.、Boeyen, S.、Housley, R.和W. Polk，“Internet X.509 公钥基础设施证书和证书撤销列表(CRL)配置文件”，RFC 5280，2008年5月。

[11] [RFC5905] Mills, D.、Martin, J.、Burbank, J.和W. Kasch，“网络时间协议版本4：协议和算法规范”，RFC 5905，2010年6月。

[12] [RFC6066] Eastlake, D.，“传输层安全(TLS)扩展：扩展定义”，RFC 6066，2011年1月。