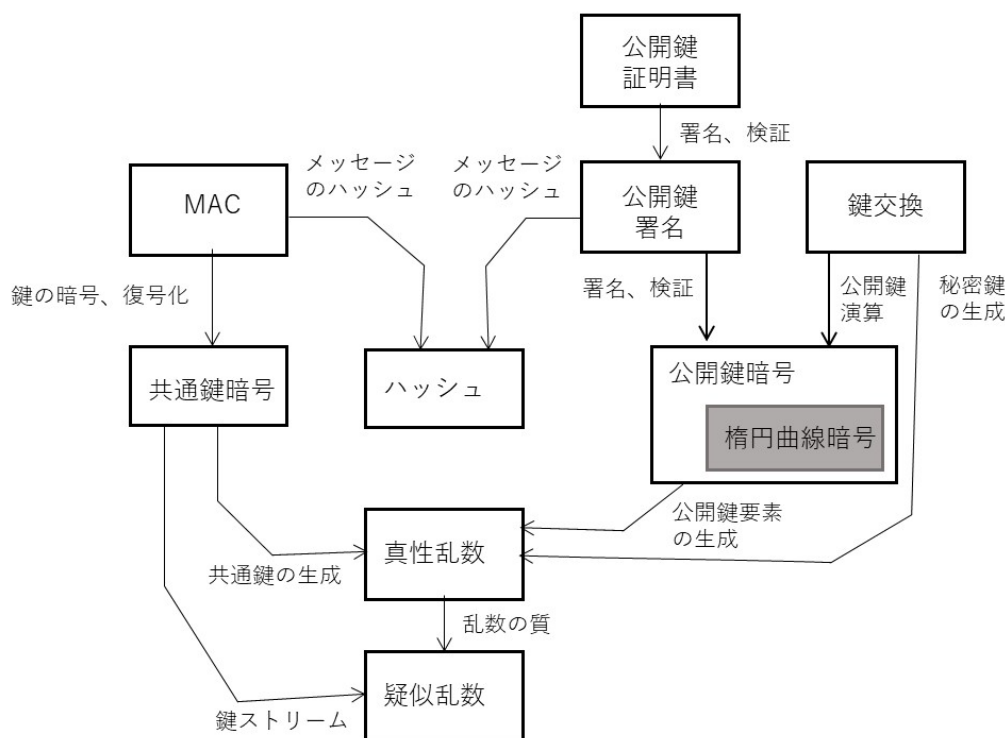


3.1 概要

情報通信技術のなかでもデジタル暗号技術は近年急激に多岐にわたり発展した基盤技術で、幅広い分野で利用されています。この章では特にTLSで使用される暗号技術、アルゴリズムについてまとめます。図1.1はそれらの暗号技術要素とその関連を示したもので、矢印は技術要素の依存関係を示します。

この図に示すように、これらの技術の多くが乱数の予測困難性に依存して組み立てられています。また、これらの暗号技術はハッシュ、共通鍵暗号、および公開鍵暗号の3つの要素技術と組み合わせた目的別の複合的な暗号技術とからできています。



1) 乱数

乱数はすべての近代暗号アルゴリズムの基本となるもので、その暗号強度は利用する乱数の質に依存しています。質の低い乱数を使用すると、例えば暗号化のための鍵の長さのようなものが十分に生かされず秘匿性が確保されないことになるので十分な注意が必要です。

真性乱数は周期性や統計的な偏りがない純粋な乱数です。質の高い真性乱数を得ることは容易ではなく、特にソフトウェアのように決定論的に動作するアルゴリズムだけでは真性乱数を得ることはできません。疑似乱数は、もともとなる乱数シード値を外から与え周期が十分に長く統計的な偏りの少ない乱数列を決定論的に生成する技術です。質の高い真性乱数を直に生成するのが難しい場合に、真性乱数をシードとして疑似乱数を組み合わせて質の高い乱数を得るというような手法が使われます。疑似乱数はまた、シード値が同じ場合同一の乱数列を生成するので、例えばシミュレーションのように再現性が必要な応用でも利用されます。

代表的な疑似乱数アルゴリズムとしては、Hash_DRBG, HMAC_DRBG, CTR_DRBGなどがあります。乱数の質に関する規定としては、例えば米国NISTによる SP 800-90A/B があります。

2) ハッシュ

ハッシュはメッセージダイジェストとも呼ばれ、不定長の長いメッセージを固定長の短いデータに圧縮するための一方性のアルゴリズムです。データを圧縮するため異なるオリジナルメッセージから同一のハッシュ値が生成される（ハッシュ衝突）可能性や、ハッシュ値から元のメッセージを推測される（原像計算）リスクがあり、使用するアルゴリズムにはそうしたリスクが最小となることが求められます。

代表的なハッシュアルゴリズムとして、初期にはMD5, SHA1が広く使用されましたが危殆化のため現在ではSHA2(SHA256/384/512など)、SHA3などが標準化され、使用されています。TLS1.3ではSHA2のSHA256/384が暗号スイートのハッシュとして採用されています。

3) 共通鍵暗号

暗号化と復号化に同じ鍵を使用する暗号アルゴリズムを共通鍵暗号（対称鍵暗号）と呼びます。共通鍵暗号は大容量のデータを効率的に暗号化、復号化できる特徴があるので、TLSではアプリケーションデータ転送時の暗号アルゴリズムとして使用します。

しかし、ネットワーク通信のように潜在的に多数の相手方と通信する可能性がある場合には使用する鍵を相手方にどのように安全にわたすかという問題（鍵の配送問題）を抱えています。TLSでは、公開鍵暗号の技術を使って通信の相手方と安全に同一の鍵を共有した上で（鍵交換、鍵合意）共通鍵暗号によって暗号通信を行います。

共通鍵暗号はブロック単位に処理をするブロック型とそのような区切れを必要としないストリーム型にわけることができます。AESは現在ブロック型として一番広く使われているアルゴリズムです。また、ブロック型の場合ブロック間の情報の接続方法によって各種の利用モードがあり、AES-CBCはTLS1.2まではもっとも広く利用されていましたが、TLS1.3でAES-GCM, AES-CCMが標準として取り入れられています。

ストリーム型のアルゴリズムとして初期にはRC4が広く使われましたが危殆化により現在は廃止されています。TLS1.3ではChaCha 20という新しいアルゴリズムが標準として取り入れられています。

4) メッセージ認証コード(MAC)

ネットワーク通信においては受け取ったメッセージが改ざんされたものでないこと(完全性)の確認が課題となります。そのためのアルゴリズム、メッセージ認証コード(MAC)にはハッシュを利用したHMAC（Hash based MAC）、共通鍵暗号を利用したAES-MACなどがあります。MACによる検証には共通鍵がつかわれ鍵の正当な所有者だけがメッセージの完全性を検証することができます。しかし、次第にメッセージ認証はより細かな単位で行う必要性が高まりメッセージ認証は共通鍵暗号の一環として取り込まれるようになってきています(AEAD)。

一方、TLSの鍵導出アルゴリズムには従来はTLS独自のアルゴリズムが使われていましたが、TLS1.3ではMACアルゴリズムの一つであるHMACが採用されています。

5) 公開鍵による鍵交換と署名

当初、公開鍵暗号（非対称鍵暗号）は共通鍵の鍵配送問題解決のために研究されました。公開鍵アルゴリズムでは、暗号化、復号化に異なる鍵を使用する暗号アルゴリズムで両者の鍵が異なり、暗号化のための鍵（公開鍵）を通信の相手方に渡して暗号化したメッセージを受け取ることができるからです。ただし、公開鍵暗号の処理は共通鍵暗号にくらべて極めて大きな処理時間がかかるため、TLSでは最初のハンドシェイクで公開鍵による鍵交換で通信の両者が同一の鍵値を得たのち、それを共通鍵暗号の鍵として使用し、大量のメッセージを効率的に暗号化、復号化するという手法がとられています。

初期のTLSではRSAによる暗号化と復号化を利用した鍵交換方式が広く使用されていましたが、RSA方式では同じ鍵を長期に使いまわす必要があるため近年は公開鍵といえども秘匿性のリスクが指摘されてきました。そのため、元となる鍵の更新が容易なディフィーヘルマン(DH)型の鍵交換が推奨され、TLS1.3ではDH型とその派生形のみが標準として認められるに至っています。

公開鍵署名は、公開鍵暗号の特性である一方向性を利用したデジタル署名のアルゴリズムです。TLSでは、公開鍵署名は通信の相手方のなりすましを防ぐピア認証の手段として使われています。通信相手としての自分の正当性を証明するためには、まず自分の公開鍵の格納された証明書（後述）とともに、自分の署名用のプライベート鍵を使用して適当なメッセージ(Blob)に対する署名を送ります。受け取った側はBlobに対する署名が正しいものかどうかを証明書に格納された公開鍵で検証します。

代表的署名アルゴリズムにはRSA, DSAがあります。RSA署名ではRSA暗号化と同様、別の鍵を使ってデータを復元できる性質(循環群)を使って署名検証を実現しますが、DSA署名では一方向演算の組合せだけで署名を検証します。このように公開鍵暗号の用途は当初の目的より大きく広がり、情報秘匿化のための暗号化技術という色合いは薄れてきています。

6) 楕円曲線暗号

楕円曲線暗号では楕円曲線上の点とそれに対するスカラー演算を定義します。RSAやDHが離散対数の逆演算困難性をベースとするのに対して、楕円曲線暗号ではスカラー値が十分大きい場合、楕円上の点に対するスカラー乗算の逆方向の演算が困難となることを暗号化に利用します。ECDHでは、この楕円曲線上の演算を利用してディフィーヘルマンに似た方法で通信する両者で安全に共通の鍵を得ることができます。楕円曲線暗号はデジタル署名のアルゴリズムとしても利用されています。楕円曲線による署名アルゴリズムとしてDSAの原理をECCに適用したECDSAが標準化されています。

一般的に、楕円曲線暗号はRSAのような素数に依拠するアルゴリズムより短い鍵で同程度の暗号強度を得ることができます。楕円曲線暗号はアルゴリズムはやや複雑になりますが、鍵が短くべき乗演算が容易なこともあり現実のアルゴリズム実行時間としては所要時間が短くなる傾向にあります。NISTは初期に楕円曲線の標準化に取り組み通称NIST曲線と呼ばれる一連の曲線を標準化しました。欧州を中心としたSECGによる曲線ともマッピングがとられTLSでも標準として採用されています。

最近では、Curve25519, Curve448のような演算処理が単純化される特別な楕円曲線も研究され、TLS1.3でも標準の曲線として採用されています。

7) 公開鍵証明書

公開鍵証明書は公開鍵とその他のメタ情報を格納したものに署名をしたものです。証明書への署名は通信の双方が信頼する認証局（CA：Certification Authority）が認証局のプライベート鍵を使用して署名します。通信を行おうとするものは、信頼するCAの証明書を使って相手から送られてきた証明書が認証局によって正当に署名された正当な証明書であることを確認できます。また、証明書に格納されている通信相手の公開鍵を使って通信相手の正当性を認証することができます。

公開鍵証明書の書式はITU X.509、またそのベースとなるデータ構造定義ASN.1、DERによって物理的構造が標準化されています。DERをBase64でテキスト形式に変換したPEM形式もファイルフォーマットとして広く利用されています。

8) 公開鍵基盤 (PKI)

公開鍵基盤 (PKI:public key infrastructure)は、公開鍵暗号、証明書などを使用した信頼できる第三者機関である認証局をベースとする信頼モデル、またそれを実現するための標準規約などです。PKIのための一連の具体的な標準と

して、公開鍵技術の初期段階からRSA社が取り組んだPKCS(Public Key Cryptography Standards)があります。PKCSはその標準のカテゴリにより番号付されていて、今でも多くのカテゴリが広い分野で標準規約として利用されています。プロトコルとしてのTLSもPKCSをベースとしたPKIの存在を前提として作られています。

3.2 乱数

1) 真性乱数

乱数は現代の暗号アルゴリズムの基本となるもので、その暗号強度は利用する乱数の質に依存しています。真性乱数を厳密に定義することは極めて難しいのですが、直観的には周期性や統計的な偏りがない予測困難なビット列、整数などの数値列というように定義することができそうです。本章では、厳密な議論ではなくTLSと関連して、乱数の扱いに関して意識すべき点を中心に解説します。

質の高い真性乱数を得ることは容易ではなく、特にソフトウェアのように決定論的に動作するアルゴリズムだけでは真性乱数を得ることはできません。

2) 疑似乱数

疑似乱数は、もとなる乱数シード値を外部から与え周期が十分に長く統計的な偏りの少ない乱数列を決定論的に生成する技術です。疑似乱数は、質の高い真性乱数を直に生成するのが難しい場合に、真性乱数をシードとして質の高い乱数を得るために使用されます。また、シード値が同じ場合同一の乱数列を生成するので、例えばシミュレーションのように再現性が必要な応用でも利用されます。TLSの中ではプレマスターシークレットからセッションで使用する種々の鍵を導出する際にも使用されます。ストリーム暗号の鍵ビット列の生成も一種の疑似乱数生成と見ることができます。

3) 乱数生成の実例

疑似乱数については、質の高い乱数生成が実現できるソフトウェアアルゴリズムが多数研究されていますが、真性乱数の生成は原理的に決定論的なロジックだけでは生成することはできません。一般的には電氣的なノイズなどなんらかのハードウェア的なエントロピー源を利用して質の高い乱数を実現します。また、疑似乱数アルゴリズムによってある程度の長さの乱数列において統計的な偏りの少ない質の高い乱数を生成することができるので、ある程度の質の真性乱数をシードとして、それをベースに疑似乱数で質を高めるという手法をとることができます。

wolfCryptの真性乱数生成でも、GenerateSeedによる真性乱数シードをベースにHash DRBGによる疑似乱数でランダム性を高める方法がとられています。Hash DRBGによって多数回の乱数生成においては統計的な偏りのない乱数系列が実現されています。しかし、当然ながら同じシード値からは同じ系列の乱数が生成されます。Hash DRBGでは、適宜の間隔でGenerateSeedにシード値を求めるような仕組みになっています。シード値の持つビット幅分の自由度をもったシード値を生成することはGenerateSeed側の責任となります。

一方、シード生成部分は真性乱数でなければなりません。真性乱数の質を厳密に定義することは難しいことです。また、質の高い真性乱数生成を自製することは難しいので、一般的にはMCUなどハードウェアやOSの提供する真性乱数生成機能の使用します。

どうしても自製しなければならない場合は、統計的な質については疑似乱数のほうで実現し、シード部分では以下のような配慮のもとに十分なエントロピーが蓄積されるように注意して実現します。

1. 最初のシード値としてシード値のビット幅分の十分な自由度をもった値が生成されるようにすること
2. シード生成ごとに異なる値を返却すること
3. 時間とともにエントロピーが蓄積されるように配慮すること

4. システムリセット時など再開の場合にもエントロピーがリセットされないように、また、前回とはことなる初期値を返却するようにすること

4) 乱数の質に関する標準

Hash DRBGなど疑似乱数としての乱数の質についてはNIST SP 800-90A の規定があります。また、真性乱数の検定に関してはNIST SP 800-90Bで検定モデルが規定されています。

<https://www.ipa.go.jp/security/jcmvp/documents/atr/atr01e.pdf>

TLSでは使用する乱数の質に関する規定はありませんが、IETFでは以下のような指針をRFCとしてドキュメント化しています。

"Randomness Requirements for Security", BCP 106, RFC 4086

3.3 ハッシュ

ハッシュはメッセージダイジェストとも呼ばれ、不定長の長いメッセージを固定長の短いデータに圧縮するための単方向性のアルゴリズムです。

ハッシュ値を求めるハッシュアルゴリズムには、ハッシュ値から逆にそのようなハッシュ値となるメッセージを得ることが（事実上）不可能であること（原像計算困難性、弱衝突耐性）ことが求められます。そのためには、メッセージをほんの少し変えたときハッシュ値は大幅に変わり、元のメッセージのハッシュ値とは相関がないように見えようなハッシュ値が得られるようなアルゴリズムであることが求められます。また、同じハッシュ値となる、異なる2つのメッセージのペアを求めることが（事実上）不可能であること（強衝突耐性）も求められます。

ハッシュアルゴリズムとして早い時期にRonald RivestによるMD5が1992年にRFC1321として標準化され、その後NISTによる標準としてハッシュビット長がより長く、大きなデータに適用できるSHA1, SHA2が標準化され広く利用されました。SHA1は160ビットのハッシュのアルゴリズムですが、SHA2は224ビットから512ビットまでのハッシュ長を得る一連のアルゴリズムの総称です。SHA2はそのハッシュ長ごとにSHA256, SHA512などとも呼ばれています。

TLSでもこれらをベースとした暗号スイートが標準として採用されてきましたが、近年MD5, SHA1に関する攻撃に関する研究が報告され、攻撃の現実化が懸念されるようになりました。そのため、MD5, SHA1はTLS1.3では完全に廃止されました。

MD5, SHA1, SHA2はMerkle–Damgård constructionアルゴリズムをベースにしており、このアルゴリズムのみへの依存に対する懸念から、新しい標準としてSHA3が制定されました。しかし、現在のところSHA2に関しては具体的なリスクは報告されておらずTLS1.3ではSHA256, SHA384が採用されています。

アルゴリズムとバリエーション		出力長 (bits)	内部状態長 (bits)	ブロック長 (bits)	最大メッセージ長 (bits)	ラウンド数	ビット演算	セキュリティ強度 (bits)
MD5		128	128 (4 × 32)	512	$2^{64} - 1$	64	And, Xor, Rot, Add (mod 2^{32}), Or	<64 (強衝突)
SHA-0		160	160 (5 × 32)	512	$2^{64} - 1$	80	And, Xor, Rot, Add (mod 2^{32}), Or	<80 (強衝突)
SHA-1		160	160 (5 × 32)	512	$2^{64} - 1$	80	And, Xor, Rot, Add (mod 2^{32}), Or	<63 (衝突発見 ^[6])
SHA-2	SHA-224	224	256 (8 × 32)	512	$2^{64} - 1$	64	And, Xor, Rot, Add (mod 2^{32}), Or, Shr	112 128
	SHA-256	256						
	SHA-384	384	512 (8 × 64)	1024	$2^{128} - 1$	80	And, Xor, Rot, Add (mod 2^{64}), Or, Shr	192 256 112 128
	SHA-512	512						
	SHA-512/224	224						
	SHA-512/256	256						
SHA-3	SHA3-224	224	1600 (5 × 5 × 64)	1152	制限なし ^[7]	24 ^[8]	And, Xor, Rot, Not	112 128 192 256
	SHA3-256	256		1088				
	SHA3-384	384		832				
	SHA3-512	512		576				
	SHAKE128	d (可変長)	1344	1088				d/2と128のいずれか小さい方 d/2と256のいずれか小さい方
	SHAKE256	d (可変長)						

表3-2 主なハッシュアルゴリズム (Wikipediaより)

3.4 共通鍵暗号

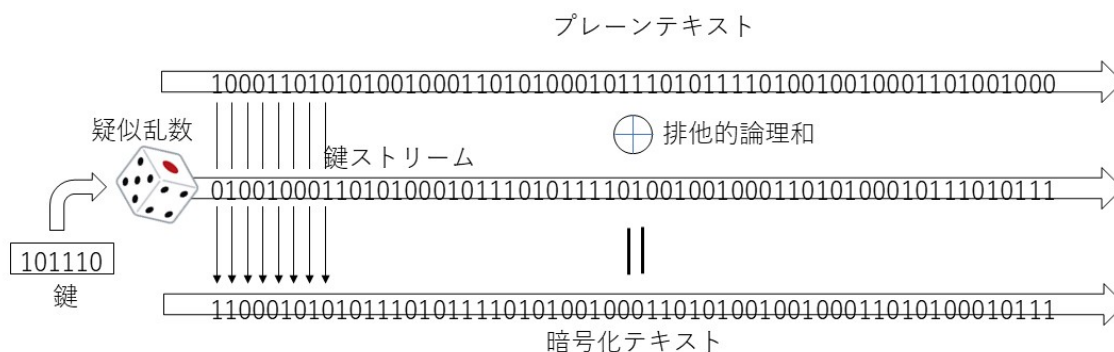
暗号化と復号化に同じ鍵を使用する暗号アルゴリズムを共通鍵暗号（対称鍵暗号）と呼びます。共通鍵暗号は大容量のデータを効率的に暗号化、復号化できる特徴があるので、TLSではアプリケーションデータ転送時の暗号アルゴリズムとして使用します。共通鍵の配送問題を解決するために、TLSでは公開鍵暗号の技術を使って通信の相手方と安全に同一の鍵を共有した上で（鍵交換、鍵合意）共通鍵暗号の処理を行います。

共通鍵暗号のアルゴリズムにはブロック型とストリーム型の二つの方式があります。ブロック型は暗号化の基本単位をブロックに分け、ブロック単位の暗号化とそれをチェーンして目的のサイズのメッセージ全体の暗号化を実現します。これに対してストリーム型は、処理の単位をわけずに処理対象全体を連続的に暗号化します。

1) ストリーム型暗号

ストリーム暗号では、与えられた特定サイズの暗号鍵を基に一種の疑似乱数生成によって暗号化しようとする平文と同じサイズのランダムなビット配列を生成します。このビット配列に対して、ビットごとに排他的論理和による暗号化演算を施すことによって暗号化を行います(図3-4-1)。このようなビット列の排他的論理和では入力となるビット列データの1/0の出現頻度と出力における出現頻度に相関関係がありません。そのためこのような構造で暗号化することで、鍵のランダム性さえ確保できればビットの出現頻度をもとに暗号文から平文を推測することを不可能とすることができます。

ストリーム暗号



ストリーム暗号は単純なアルゴリズムで処理を実現できるという大きな特徴を持っていますが、その暗号強度は疑似乱数生成の質の如何にかかっています。ロナルド・リベストにより初期に開発されたRC4はTLSやその他のプロトコルにおいて広く使用されていましたが、近年、RC4に対する攻撃法が報告されるに至り、現在では使用されなくなっています。一方、その後ダニエル・バーンスタインによって開発発表されたSalsa20はChaCha20(RFC 7539)として改良されました。ChaCha20はメッセージ認証符号Poly1305との組み合わせで認証付き暗号を実現し(「3) 認証付き暗号」参照)、現在のTLSで共通鍵暗号アルゴリズムの一つとして採用されています。

2) ブロック型暗号と利用モード

ブロック型暗号は、メッセージを固定長のブロックにわけてブロックごとに暗号化、復号化の処理をします。1ブロックの暗号化アルゴリズムとして初期に開発されたDES(Data Encryption Standard)の鍵長の制約を改善したトリプルDESはTLSの初期の暗号アルゴリズムとして使用されました。その後、コンピュータ処理でより効率的に処理できるアルゴリズムとしてAES(Advanced Encryption Standard)が開発され、ブロック長さ128ビット、鍵長128, 192, 256ビットのものが標準化され、現在まで広く使われています。

ブロック暗号では、必要なメッセージサイズの処理をするためにブロックを複数つなぎ合わせて処理をします。ブロックをつなぐ方式についてもさまざまな方式が提案されています。最も単純なものはECB(Electronic Codebook)と呼ばれ、すべてのブロックで同じ鍵とIVを使用しますが、これではメッセージの秘匿性に限界があります。前のブロックの暗号メッセージと次のブロックの平文メッセージの排他的論理和をブロックのIVとするCBCモードは比較的に簡単なアルゴリズムで高い秘匿性を実現できるので、AESブロック暗号と組み合わせたAES-CBCモードは最近までTLSで広く使われてきました。

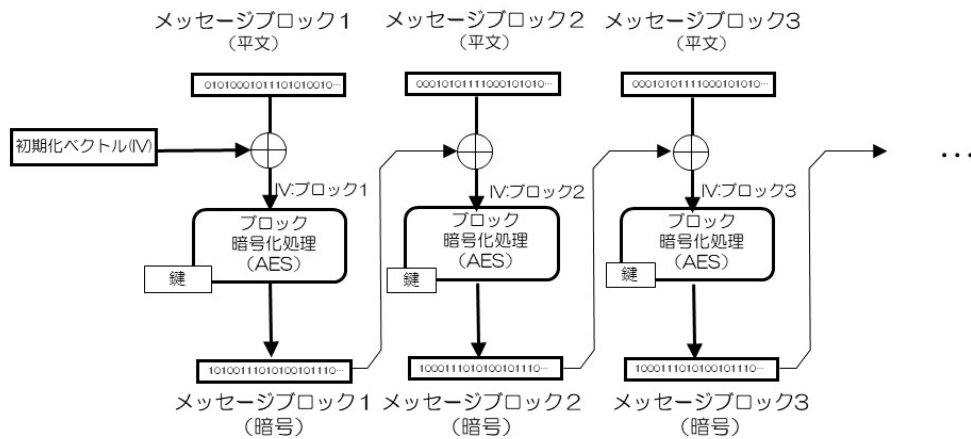


図3-4-2: 暗号ブロック・チェーン(CBC)

CTRモードは図3-4-2のように、全体のIVとして上位に適切なnonce値、下位をバイナリ整数ゼロを与え、ブロックごとにこれを1ずつインクリメントさせブロックごとのIVとして可変長の鍵を生成します。平文メッセージをこの可変長の鍵と排他的論理和をとることで暗号化します。CTRモードAESのようなブロック暗号を要素アルゴリズムとしていますが、このような形で可変長の鍵を生成し暗号化するので実質的にストリーム暗号とも言えます。

CTRモードは簡単なアルゴリズムでありながら秘匿性をそこなわず、またブロックごとの依存性はカウンターだけなのでブロック番号さえわかればブロックごとに並列に処理に適しているという特徴もっています。

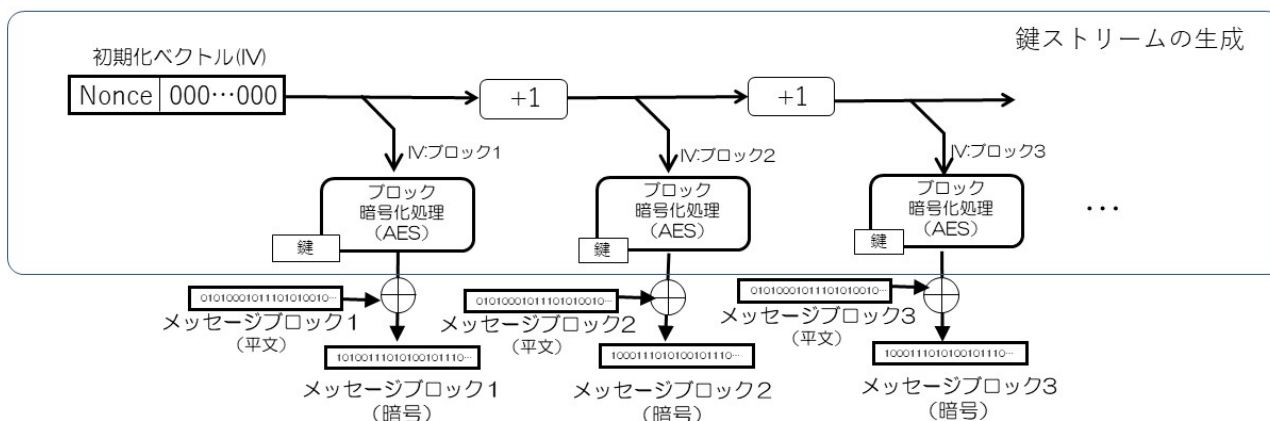


図3-4-3: カウンターモード(CTR)

3) 認証付き暗号(AEAD: authenticated encryption with associated data)

この種の暗号化アルゴリズムでは、復号化ができたからと言っても元の平文が改ざんされていないこと(メッセージの真正性)の保証にはなりません。このためTLSでは、暗号化とは別にレコード毎にメッセージ認証コード(MAC: Message Authentication Code)を付与して真正性の検証をしていました。しかし、近年この方法では真正性を完全に保証できないリスクが指摘され、TLS1.3では暗号化処理と組み合わせでより細かな単位でメッセージ認証ができるAEAD方式のみが採用されることになりました。そのため、CTRモードは次に説明する認証付き暗号を実現するためのベースの仕組みとして利用が継続されているものの、単体のCBCモード、CTRモードは標準から除外されています。

認証付き暗号はメッセージの暗号化、復号化と同時にメッセージの真正性の確認も行う暗号アルゴリズムです。真正性のチェックに使用する認証タグをメッセージの暗号化処理と同時に生成し、復号化処理の際にその認証タグによって真正性をチェックします。TLS1.3で使用されている認証付き暗号としては、ブロック型のAES-GCM(Galois/Counter Mode)、AES-CCM(Counter with CBC-MAC)およびAES-CCM_8、ストリーム型のChaCha20-Poly1305があります。

AES-GCMでは、暗号処理部にAES-CTR、認証タグの生成、認証にGMAC(Galois Message Authentication Code)を使用して認証タグ付き暗号を実現しています。図3-4-3に暗号化処理の場合のアルゴリズムの流れを示します。GMAC処理部は認証データ値を入力として、暗号化されたメッセージから認証タグ値を導きます。この認証タグ値は復号化処理の入力として真正性のチェックに使用されます。

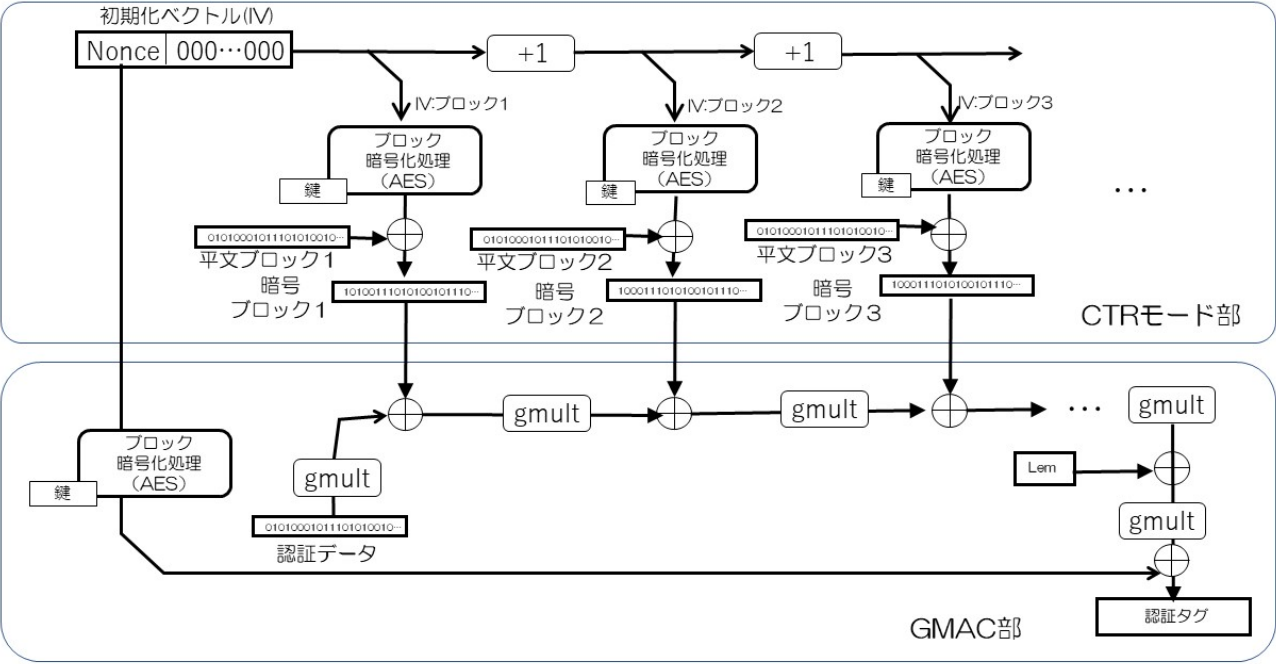


図3-4-4: GCMモード(暗号化)

AES-CCM、AES-CCM_8も暗号化部分には同じくAES-CTRを使用しますが、認証タグはCBC-MACを使用します。CBC-MACはGCMのGMACより処理が軽いため、これらの利用モードは比較的低い組込みシステム向けなどに利用されています。

表3-4-1 に廃止となったものを含めてTLSの主な共通鍵暗号をまとめます。

方式	アルゴリズム	利用モード	鍵長(Bit)	TLS1.2以前	TLS1.3	備考
ブロック暗号						
	Blowfish	-	32~448	✓		
	Camellia	CTR/CCM	128/192/256	✓		RFC3713
	DES3	各種	168	✓		
	AES	ECB	128/192/256			
		CBC		✓		RFC3268
		CTR		✓		
		GCM		✓	✓	
		CCM		✓	✓	RFC6655
		CCM_8			✓	
ストリーム暗号						
	RC4	-	40~256	✓		RFC2246
	Chacha20	-	256	✓	✓	RFC7539

表3-4-1 TLSの主な共通鍵暗号

3.5 鍵導出

鍵導出は、一定サイズのビット列から用途に応じたサイズのビット列(疑似乱数値)を生成するためのアルゴリズムです。ストリームのように長いビット列を小さなサイズのビット列に圧縮するために用いる場合と、元となるビット列より長いビット列に拡張するために用いる場合があります。前者はハッシュと同じように機能し、後者は疑似乱数生成と同じように機能します。単なるハッシュや疑似乱数と異なり、共有鍵暗号のメカニズムを併用することで鍵を知っている正当な当事者だけが正しい値を導出できるようになっています。

TLSでは鍵共有プロトコルによって通信ノードの両者が同じプレマスタークレット値を共有した後、その値をもとにアプリケーションデータの暗号、復号のための共有鍵やIVを得るために利用しています。TLS1.2まではTLS専用のPRF(Pseudo Random Function)が使用されていましたが、TLS1.3では他のプロトコルでも共通に使用されているHKDF(HMAC-based Extract-and-Expand Key Derivation Function)が使用されています。HMACはハッシュとAESによる共通鍵を組み合わせたアルゴリズムです。TLS1.3ではSHA256とSHA383によるHMACが標準として規定されています。

3.6 公開鍵暗号と鍵共有

3.6.1 背景

公開鍵暗号は、当初は暗号化と復号化に異なる鍵を使うことから片方の鍵を公開できるようにした暗号方式として研究されました。しかし今日、公開鍵のもつ性質はさまざまな局面で応用され、その適用分野は多岐にわたっています。当初の情報秘匿のための暗号化、復号化を目的とした利用方法はむしろ主要の目的とは言えない状況となりつつあります。

TLSの中でも、公開鍵暗号の技術は主に共通鍵の鍵配送問題の解決のための鍵交換ないし鍵共有、あるいは公開鍵署名とそれをベースにした証明書として使用されています。本章では、公開鍵の基本的な考え方とそれを利用した鍵交換、デジタル署名を中心に説明します。証明書については、3.8 公開鍵証明書 で説明します。

RSA暗号はロナルド・リベスト、アディ・シャミア、レオナルド・エーデルマンにより発明された代表的な公開鍵暗号で、離散対数演算の困難性に基いて秘匿性を実現した暗号方式です。さらにこのとき、適切な素数の組 (e, d, n) を選ぶと、下のように平文メッセージ m を e 乗し n で割ったあまりが暗号化メッセージ c となり、 c を d 乗し同様に剰余をとることでもとの復号化メッセージ m を得ることができます。RSA暗号では、このような条件の成立する数値の組を利用して、整数値 e, n を暗号化のための鍵、 d, n を復号化のための秘密鍵として使います。

RSAで使用している指数の剰余演算の逆演算は離散対数演算です。この離散対数演算は数値が十分に大きな素数であれば演算が極めて困難で簡単な計算方法は見つかっていません。つまり、指数の剰余演算は実質的に一方向のみ可能な演算なので、暗号化の鍵は公開しても暗号化した情報を秘匿することができます。また、適切な数値ペアを選ぶことで元の数値に戻ることができる循環演算が存在することで暗号化と異なる鍵で復号化することが可能となります。

- 暗号化（平文 m から暗号文 c を作成する）： $c = m^e \bmod n$
- 復号（暗号文 c から元の平文 m を得る）： $m = c^d \bmod n$

3.6.2 RSAの実用化技術

1) 実用的な剰余演算

この原理のままでは計算途上で巨大な整数を扱うことになり、実用的ではありません。これを削減するためのアルゴリズムも知られていて、それらを利用することで鍵サイズの2倍の領域があればこのような計算が可能となります。例えば、中国の剰余定理に基づく削減方法やモンゴメリ削減などがあります。wolfSSLライブラリーではモンゴメリ削減を使用しています。

2) 確率的素数

RSAのようなアルゴリズムで逆演算を困難とするためには大きな素数を使用する必要がありますが、単純な素数生成アルゴリズムでは大きな素数を見つけるには長い処理時間がかかってしまいます。そのため実用的な暗号ソフトウェアの多くでは確率的素数判定法が取り入れられています。確率的素数判定では一定の確率で素数でない数値を許してしまいますが、その確率が十分に低いことで実用上暗号の秘匿性をそこなわないようにすることができます。

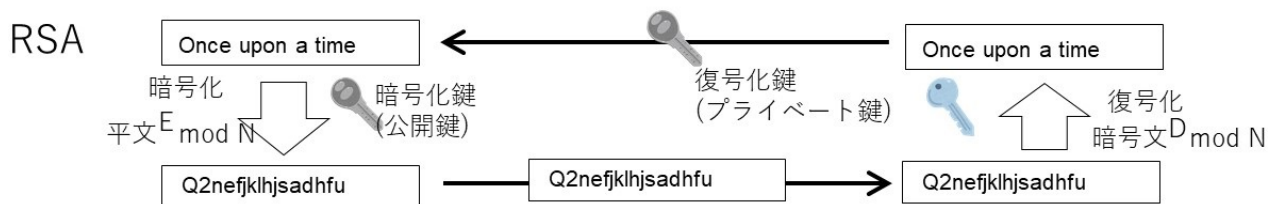
3) パディング

公開鍵暗号は復号ができたからと言って、そもそも対象メッセージが改ざんされたものでないこと、正しいものである保証（真正性の保証）はありません。もとのメッセージに付加的なパディングを挿入することでこれを検証する方法が開発されています。RSAにおけるパディングのために基本的な暗号アルゴリズムを組み合わせたパディングスキームについては、当初RSA社によりPKCS#1にて定義されましたが、現在はIETFに引き継がれRFC8017の中で規定されています（詳細は3.6.6 公開鍵暗号に関する標準を参照）。

3.6.3 RSAによる初期の鍵交換

RSAによる公開鍵暗号はTLSの初期には共通鍵暗号の鍵配送問題を解決するための鍵交換プロトコルとして広く利用されました(図3-6-1)。まず、暗号化メッセージを受け取りたい受信側は送信側に対して公開鍵を暗号化用の鍵として送付します。メッセージの送信側は受け取った公開鍵で送信メッセージを暗号化し受信側に送ります。TLSの場合、

その後の共通鍵暗号によるアプリケーションメッセージの秘匿化に使用する鍵の元となるプレマスターシークレットを送ります。受け取った側はプライベート鍵のほうを利用してこれを復号化します。



TLSの利用シナリオを考えると、サーバの成りすましを防止するためのサーバ認証も行う必要があります。当初は、上記の公開鍵の送付の際に単体の公開鍵を送るのではなくサーバ認証のためのサーバ証明書を送るようにすればその中に含まれている公開鍵がそのまま利用できるのが都合であると考えられ、TLSの初期にはそのような利用方法が標準化されました。

しかし、時代とともにセキュリティリスクも変化し、同じ公開鍵を長期に使い続けること(静的公開鍵)のリスクが指摘されるようになってきました(五章 安全性、脆弱性：完全前方秘匿性参照)。このリスクを回避するためには鍵ペアを頻繁に更新する必要がありますが、証明書の場合、認証局の署名を頻繁に更新するのは現実的ではありません。

またその間、暗号アルゴリズムの進歩も著しく、鍵交換に利用するアルゴリズムの選択と証明書のアルゴリズムは独立の選択基準で選択したいという要求も強くなってきて、サーバ認証のための証明書の送付と鍵交換のためのプロトコルは独立させたほうがよいという認識が強くなりました。

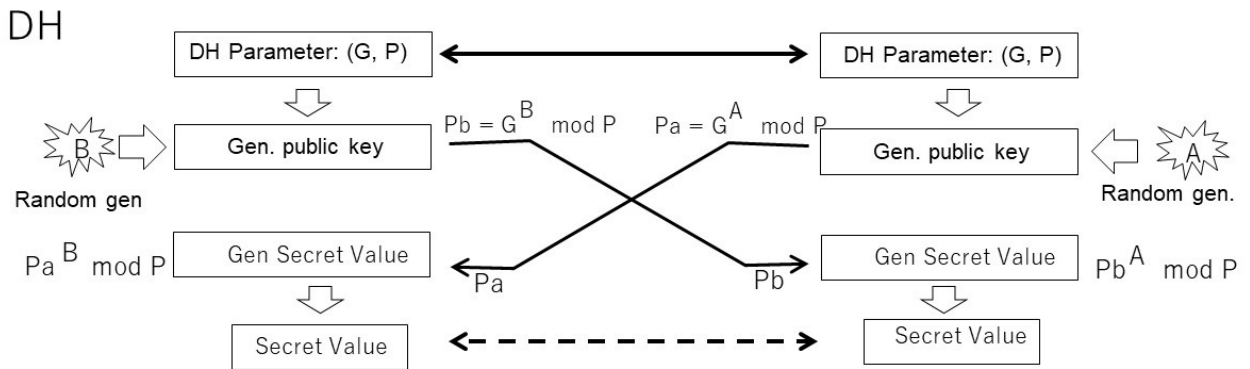
そのような背景で、TLS1.2の時代にはRSAの静的公開鍵による鍵交換は推奨されなくなりTLS1.3では廃止され、RSAによる公開鍵アルゴリズムは証明書用(八章 公開鍵証明書参照)に限定されるようになりました。

3.6.4 ディフィーヘルマン鍵交換

RSAとほぼ同時期にもう一つの公開鍵アルゴリズム、ディフィーヘルマン鍵交換(DH: Diffie-Hellman key exchange)が発明されました。DHは、RSAのように暗号化したデータを復号化するようなことはできませんが通信しようとする2者の間で共通の値を得ることができます。これを鍵交換 (鍵合意)に利用することで、鍵配送問題を解決することができます(図3-6-2)。

DHでは、RSAとは異なり離散対数演算の一方方向性とともに二回の演算の順序の可換性を利用します。具体的には、次のような手順で鍵交換を実現します。

まず、共通の鍵値を得ようとする両者は、はじめに共通のパラメータである一組の素数(DHパラメータ)を共有しておきます。このパラメータは第三者に公開することができる値です。



鍵を交換するために、両者はそれぞれ相手や第三者に対して秘密の乱数値(秘密鍵)を生成します。この値に対して先ほどのDHパラメータを使ってべき乗の剰余を求め、その値を相手側にわたします。これは前述のRSAの公開鍵による暗号化と同様に暗号化した値からもとの値を知ることは困難な一方演算なので、公開の値(DH公開鍵)として相手方に渡すことができます。

受け取った側では、この値と自分の秘密鍵、DHパラメータを使い最終的な共有鍵の値を求めます。両者の演算内容を比べてみると、単に演算の順序が異なるだけで演算の構造は同じとなっていることがわかります。

この二段階のべき乗の剰余演算が可換であることは別途証明できるので、両者の秘密鍵がどのような値であってこのアルゴリズムによって共通の値を得ることが保証できます。TLSではこの値をプレマスターシークレット（その後の共通鍵暗号で使用する鍵、IVなどを求める元の値）として使用します。

TLSハンドシェイクの利用シナリオにおけるこれらのパラメータ値や公開鍵の送付方法はTLS1.2までとTLS1.3ではやや異なります。

TLS1.2までは、ClientHelloとServerHelloは使用する暗号スイートの合意のために限定され、実際にDHで使用するDHパラメータやDH公開鍵は2往復目のClientKeyExchangeとServerKeyExchangeによって送られていました。

TLS1.3ではハンドシェイクが整理され、DHパラメータや公開鍵はClientHelloとServerHelloのKeyShare拡張に格納されるようになり、サーバはClientHelloの受信内容、クライアントはServerHelloの受信内容と自分の秘密鍵を使ってプレマスターシークレットを得て、そこからセッション鍵を導出(3.5：鍵導出参照)できるようになりました。これによってTLS1.3ではハンドシェイクを1往復で完了することができるようになったのと同時に、ハンドシェイクの途中から内容を暗号化することも可能となりました。

3.6.5 デジタル署名

デジタル署名（公開鍵署名）は、メッセージの正当性の確認のために使用します。また、デジタル署名の署名は正当な署名者以外は正当な署名が生成できないので、署名者の確認、逆に署名を生成したことの否認防止のためにも使用できます。

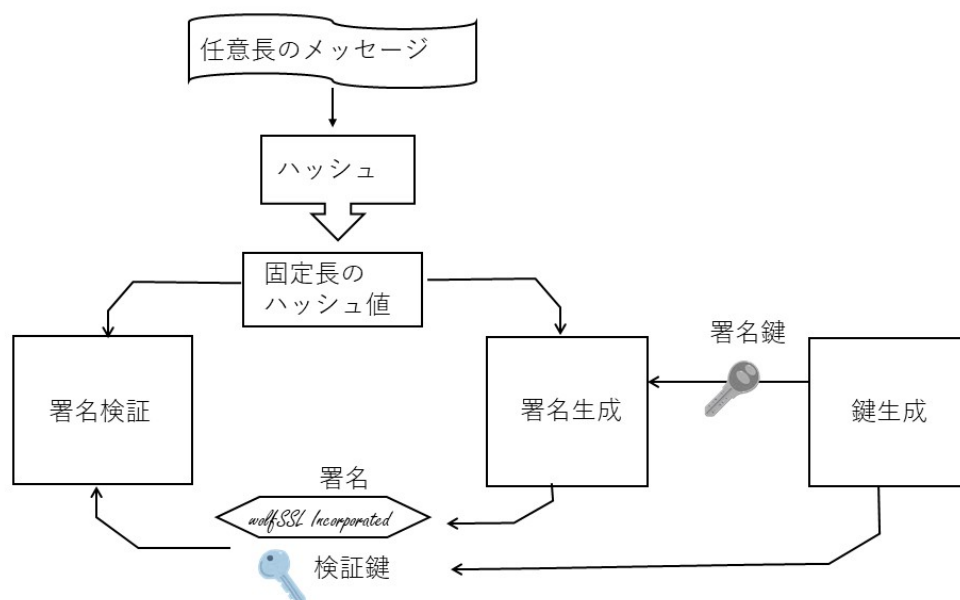
メッセージと署名の対応関係の検証だけであれば共通鍵によるメッセージ認証コード(MAC)でも可能です。しかし、MACでは鍵を知っている署名の検証者自身も正当な署名を生成することができてしまうので、正当な署名者の確認や署名否認の防止のためには利用できません。

公開鍵による署名では、署名生成鍵と検証鍵が異なるので秘密鍵を持ったものだけが署名可能です。そのため、署名の正当性を確認することでその署名が正当な署名者によるものであること確認することができます。また、逆に署名

したことの否認防止の目的にも利用することができます。

図3-6-3にデジタル署名の構造を示します。デジタル署名では、任意長のメッセージに署名することできるように、まず対象メッセージの固定長のハッシュ値をもとめます。署名の生成では、このハッシュ値と署名者だけが知る何等かの秘密の値（署名鍵）を使用して署名を生成します。一方署名の検証では、ハッシュ値、署名と署名検証用の鍵をもとに署名の正当性を検証します。

デジタル署名のしくみ

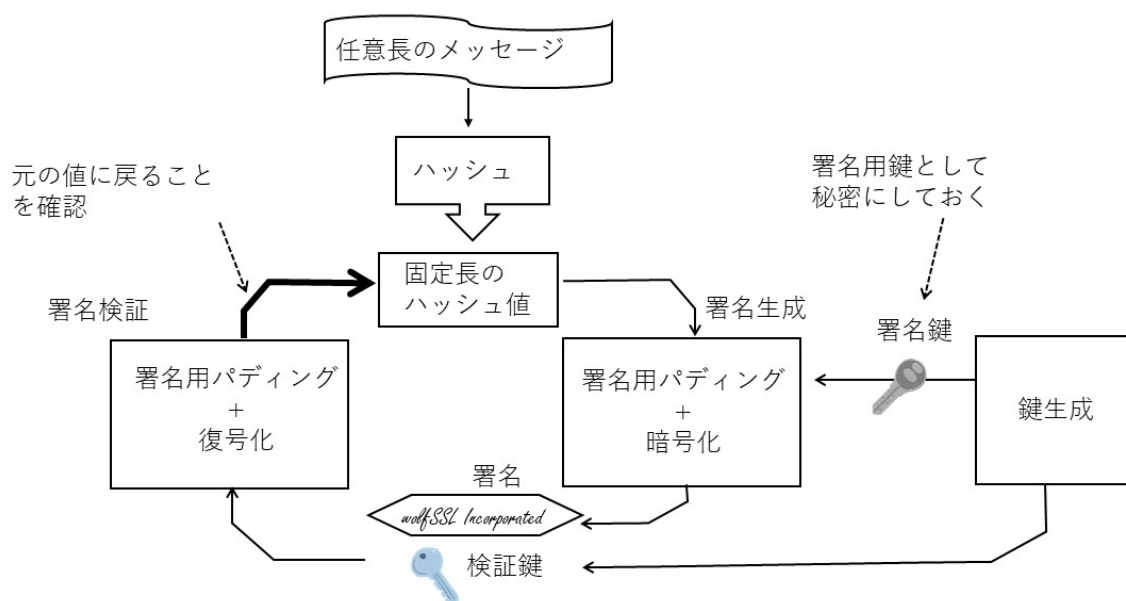


1) RSA署名

RSA署名では、RSA演算によって暗号化したものが復号化でもとに戻ること（巡回群）を利用して署名の生成、検証を実現します。署名の生成はRSA暗号化のアルゴリズムの際の公開鍵に相当する鍵と暗号化に相当するアルゴリズムを使って、メッセージのハッシュ値から署名を生成します。

一方署名の検証は、メッセージのハッシュ値と署名、署名検証用の鍵を利用します。検証用の鍵は暗号化の際の復号用のプライベート鍵に相当します。署名検証鍵によって元のハッシュ値が得られれば、メッセージと署名は正当なものであることが検証できたことになります。

RSA署名のしくみ

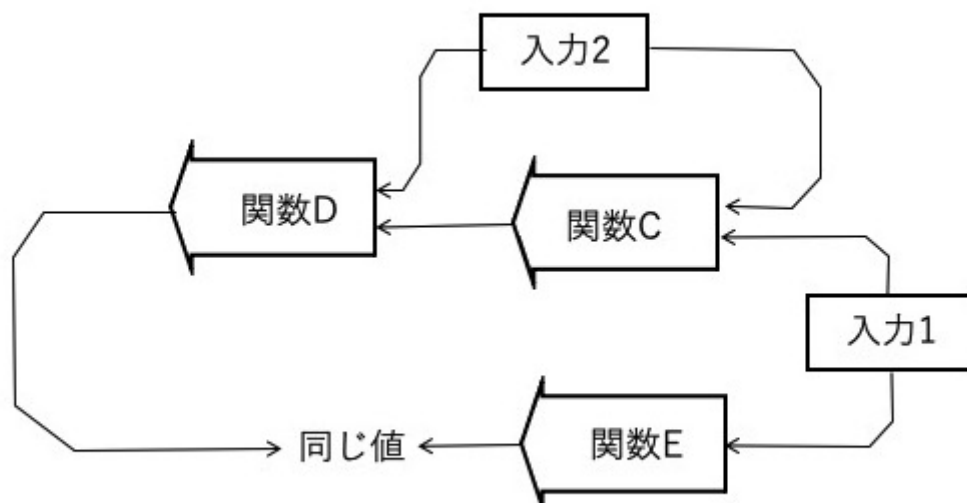


RSA署名では、署名用の鍵は署名者だけの秘密とし、検証用の鍵は検証用として公開します。暗号、復号の際とは公開、非公開としては逆の使い方をすることで、原理的にはRSA暗号、復号と同様のアルゴリズムを使用して実現することができます。しかし、今日の実用的なRSA署名として標準化されている署名スキームでは、パディングスキームは暗号復号用のものとは異なり、互いに流用することはできません。

2) DSA署名

RSA署名では巡回群を使ってメッセージのハッシュをエンコーディングして署名を生成し、検証の際はそれを逆にデコードして確認する、という方法がとられていますが、SA(Digital Signature Algorithm)署名ではそのような巡回群は利用しません。DSAでは、二つの異なる一方向演算の組み合わせ同士で同一の値を得ることができることを利用します。

DSAの原理について理解するために一旦署名と言うことから離れ、まずは図のように関数CとDの二段階の関数で、関数CとDにはもう一つの入力「入力2」も受け取るような関数を考えます。このような関数の組み合わせで関数Cと関数Eに同じ入力1を、そして関数CとDに同じ入力2を加えると関数DとEが同じ結果となるような関数を作り出します。



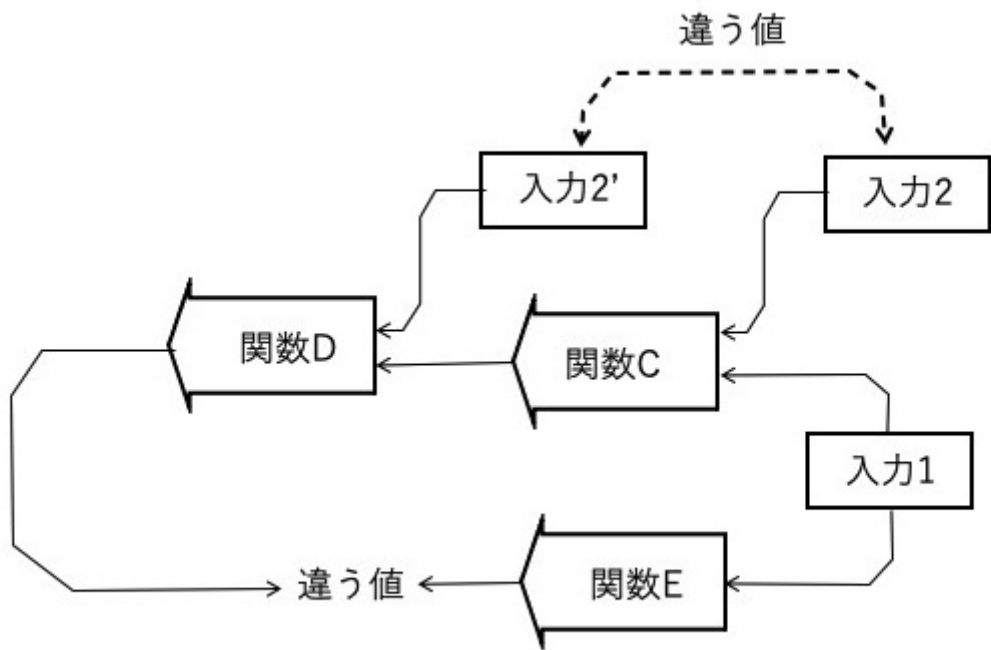
簡単のため「一方向関数」という条件をはずして考えます。入力1を x 、入力2を y とすれば、例えば次のような関数が考えられます。

関数C: $x + y$ 、 関数D: $(x - y)^2$

関数E: x^2

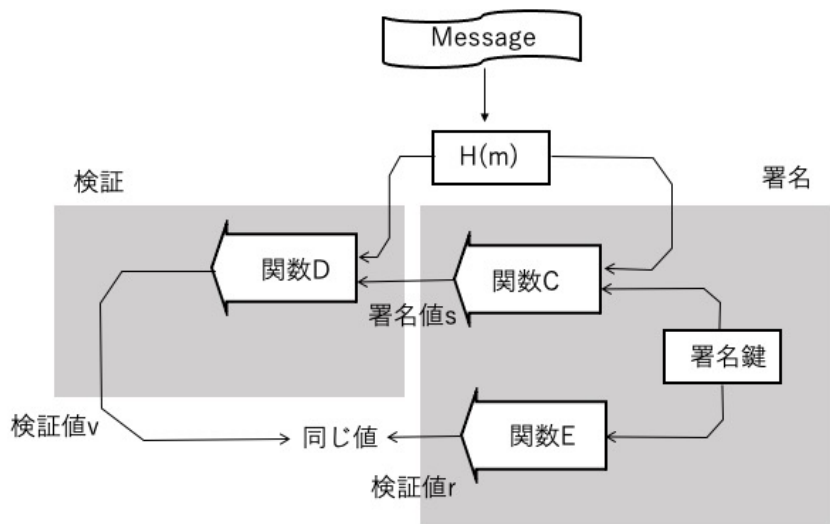
関数cで足された y は関数Dで引かれるので、値は当然もとにもどります。つまり、関数CとDの中に入力2を打ち消すような演算を入れておけば、上の関数は結局下の関数と同じになるということがわかります。

前の図ではC、Dとも同じ入力2が与えられていましたが、次の図のようにもし仮にC,Dの入力2に異なる値が与えられた場合はうまく打ち消しあわないことになります。関数Dと関数Eの結果は当然違う値になるはずです。

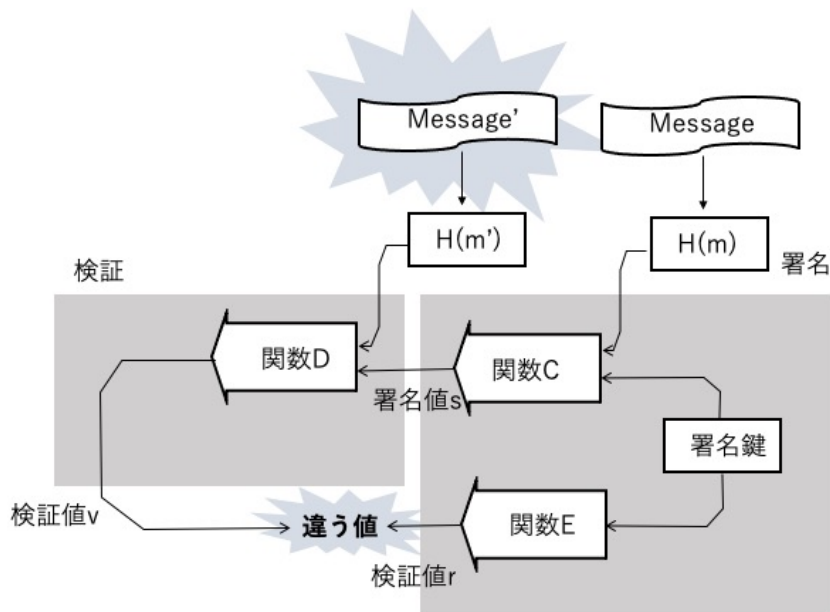


これを署名検証の流れに当てはめてみます。前の図の入力1は署名のための鍵です。入力2は署名対象のメッセージのハッシュ値です。DSAでは署名は署名値sと検証値rの二つの値の組み合わせとなります。関数Cの結果が署名値s、関数Eの結果が検証値rです。このよう当てはめると下の図のようになって、図の右半分が署名、左半分が検証の流れとなっていることがわかります。

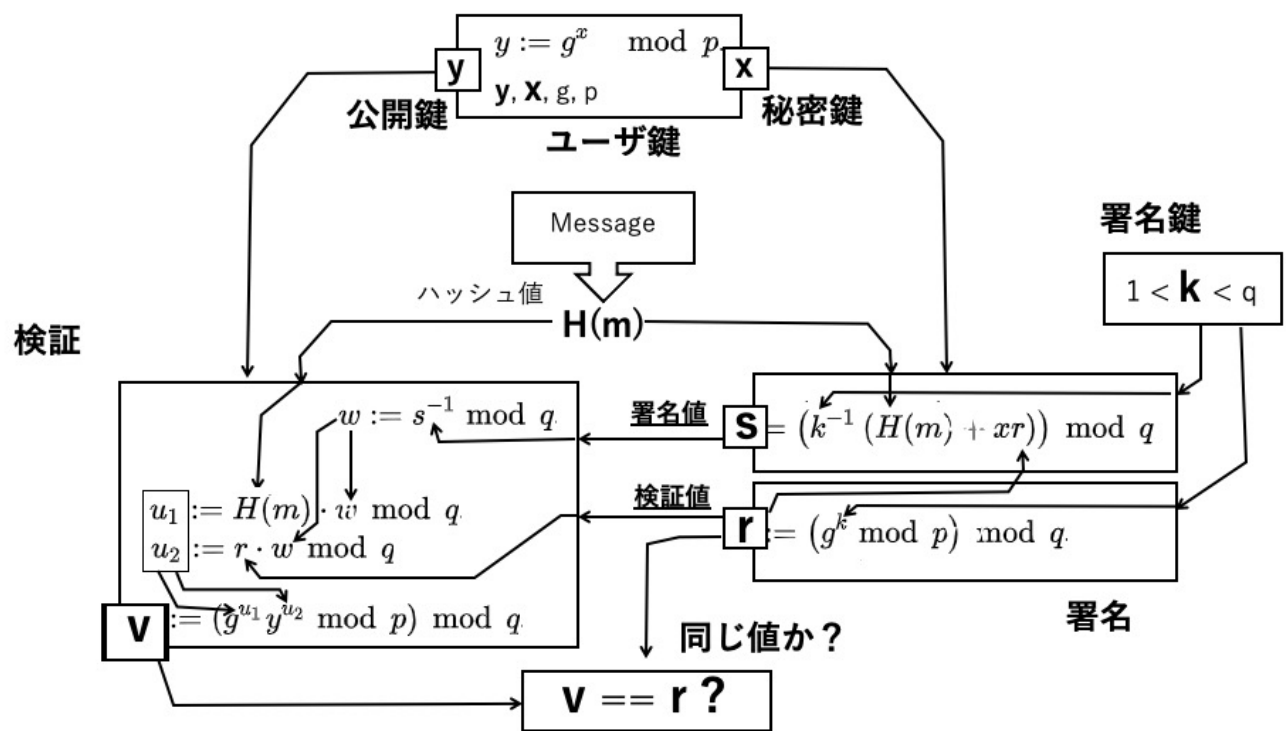
署名側と検証側でH(m)が同じ値ならば、検証値vとrは同じ値となります。



もし、例えばメッセージが改ざんされていて検証側のH(m)が署名側と異なれば検証値は一致しません。これで、署名の正当性が検証できたことになります。



最後に、この図にDSAの実際の演算式を当てはめると下の図のようになります。



一方向演算を実現するためのパラメータとしてユーザ鍵ペアが加えられていたり、署名値 r も検証演算に加えられていたり詳細の追加はありますが、基本的な構造は前の図と変わらないことがわかります。数学的厳密な意味は別として、随所に整数による公開鍵演算のプリミティブである冪乗の剰余が使われ一方向性が実現されていることを直感的に感ることができるかと思います(modは剰余を表します)。

DSAでは、このような形の巡回群の性質を利用しない演算構造で署名、検証を実現しています。

DSAは実現においては適切な鍵生成が難しく十分な注意が必要です。鍵を解読されないためには署名毎に新しい異なる鍵を生成する必要があります。また、検証のための計算量はRSAくらべてかなり大きくなる傾向にあります。そうした理由で、整数演算の世界においてはRSA署名のほうが広く利用されています。

しかし、楕円曲線暗号では巡回群は見つかっていないため、RSAのような方法で署名検証を実現することはできません。整数演算の世界のDSAと等価のアルゴリズムを楕円曲線暗号の世界で実現したECDSA, EdDSAなどが広く使われています。

3.6.6 公開鍵暗号に関する標準

1) PKCS #1: RSA暗号

基本的なRSA暗号に関する標準規定は当初PKCS #1として制定されましたが、現在ではこの内容はIETFのRFCに引き継がれています。2021年時点での最新であるPKCS #1 V2.2 (RSA Cryptography Specifications Version 2.2)はRFC 8017として定義されていて、暗号、復号、署名と検証の方法（プリミティブとスキーム）などの規定が含まれています。

分類	パディング	略称	機能	説明
鍵タイプ			公開鍵	公開鍵の基本要素(n, e)
			秘密鍵形式1	簡易秘密鍵の基本要素(n, d)
			秘密鍵形式2	秘密鍵の基本要素(p, q, dP, dQ, qInv)
データ変換プリミティブ		I2OSP	整数八進プリミティブ	整数から八進変換
		OS2IP	八進整数プリミティブ	八進から整数変換
暗号プリミティブ		RSAEP	暗号化プリミティブ	公開鍵によるパディング無暗号化
		RSADP	復号化プリミティブ	秘密鍵形式1、2によるパディング無復号化
		RSASP1	署名プリミティブ	秘密鍵によるパディング無署名
		RSAPV1	検証プリミティブ	公開鍵によるパディング無検証
暗号スキーム	OAEP	RSAES-OAEP	暗号化オペレーション	OAEPパディング公開鍵による暗号化
			復号化オペレーション	OAEPパディング秘密鍵による復号化
	v1.5	RSAES-PKCS1-v1_5	暗号化オペレーション	v1.5パディング公開鍵による暗号化
			復号化オペレーション	v1.5パディング秘密鍵による復号化
メッセージ署名スキーム	PSS	RSAES-PSS	署名オペレーション	PSSパディング秘密鍵による署名
			検証オペレーション	PSSパディング公開鍵による検証
	v1.5	RSAES-PKCS1-v1_5	署名オペレーション	v1.5パディング秘密鍵による署名
			検証オペレーション	v1.5パディング公開鍵による検証
エンコード方式	PSS	EMSA-PSS	エンコードオペレーション	PSSパディング
			検証オペレーション	PSSパディングの検証
	v1.5	EMSA-PKCS1-v1_5	エンコードオペレーション	v1.5パディング

表3-6-1 PKCS#1 (RFC8017)のRSA公開鍵スキーム

ハッシュ 備考

SHA1

表3-6-3 マスク生成関数(MGF1)のハッシュオプション一覧

この標準規定の中には、RSAのためのパディングスキームについても規定されています。当初、PKCS#1 v1.5として比較的単純なスキームが規定されましたが、その後より改善された方式として、暗号復号化には最適非対称暗号パディング(OAEP: Optimal asymmetric encryption padding)、また、RSAを公開鍵署名のために利用する場合のパデ

イングとしては確率的署名スキーム(PSS: Probabilistic signature scheme)が標準化されています。PKCS#1 v1.5も後方互換のために残されていますが、現在はこれらを使用することが推奨されています。

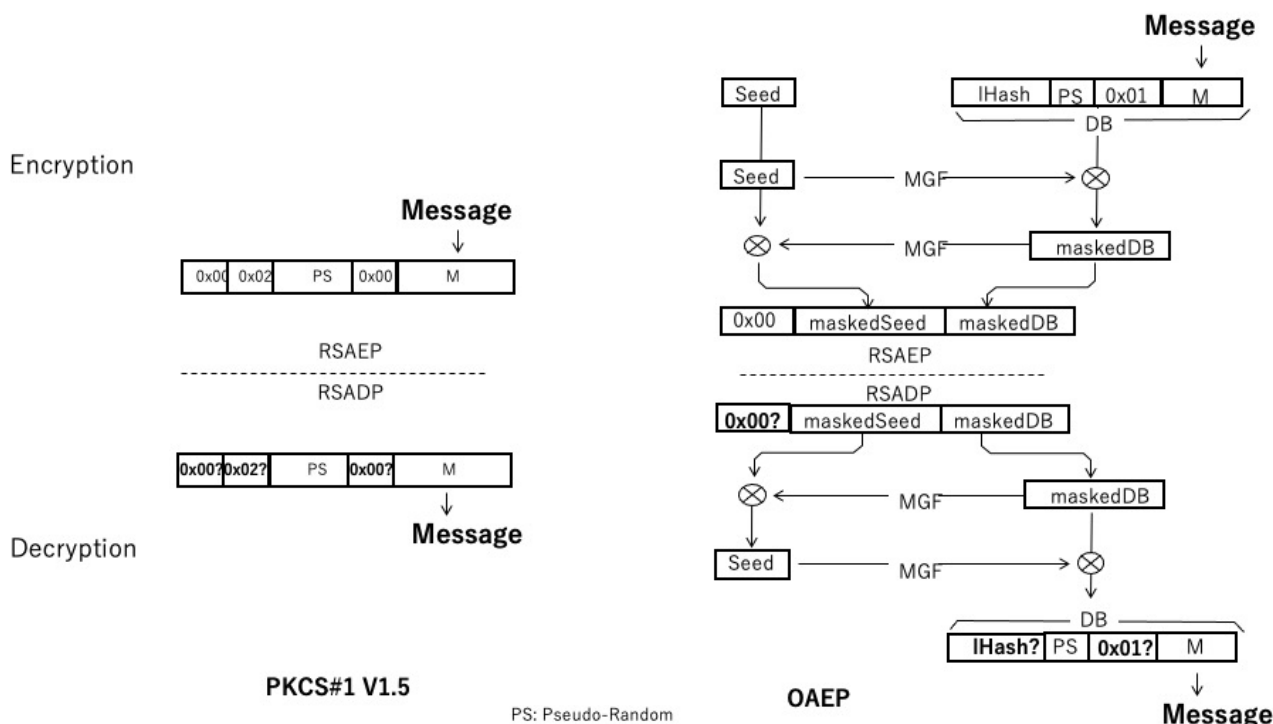


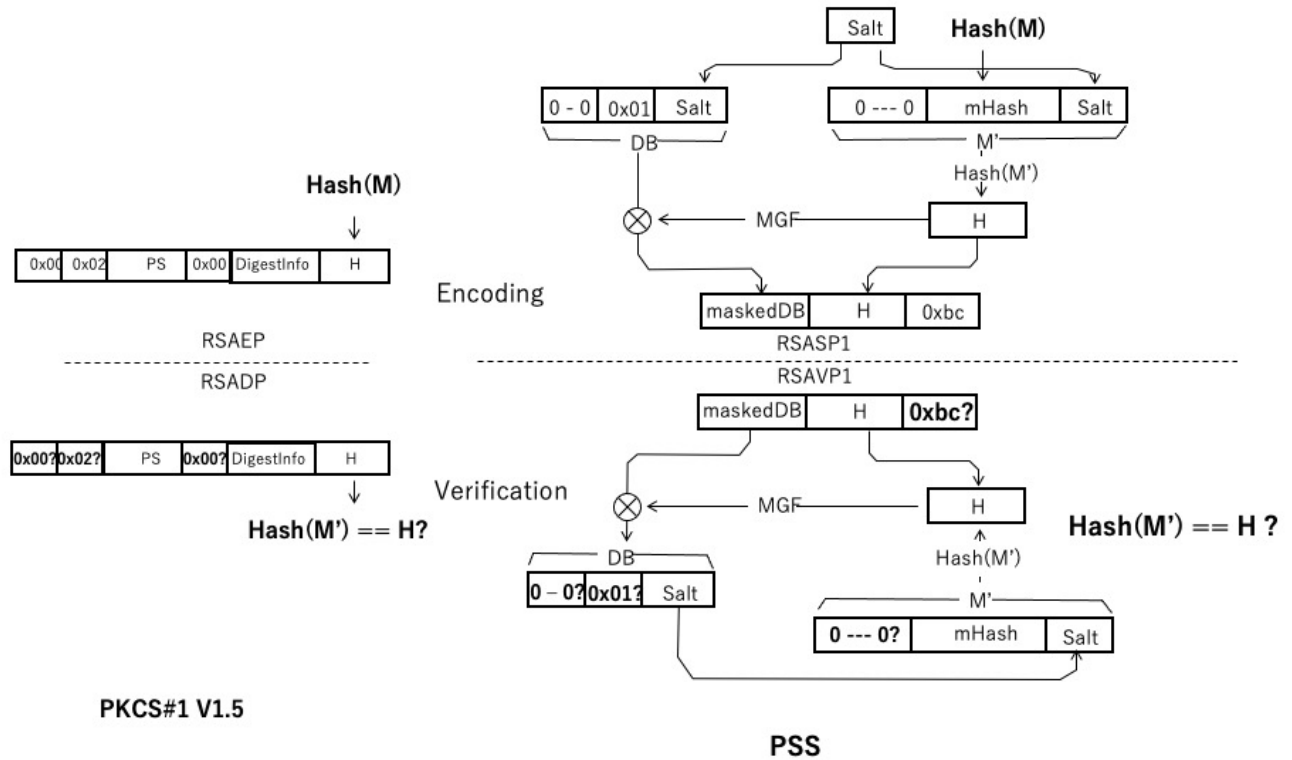
図3-6-6にPKCS#1 v1.5とOAEPのパディングスキームを比較します。

v1.5のスキームでは、暗号化対象のメッセージに所定の固定のビットパターンとハッシュ関数による擬似乱数で構成されたパディングを付加して、全体をあわせてRSA暗号化プリミティブで暗号化します。このような暗号化を行うと、パディングの擬似乱数値を知らずに固定パターン部分と元のメッセージを捏造することは非常に難しくなります。復号の際は、パディング部分についてもとの固定パターンが正しく復号されていることを確認することで元のメッセージの真正性を判定します。

OAEPでも、暗号化対象のメッセージにパディングを付加します。図においてIHashは使用するハッシュアルゴリズムによって決まる固定値、PSは擬似乱数値です。OAEPでは、これとは別に適当なSeed値を用意します。暗号化の前に図に示すようにこの値とふたつのハッシュ関数を使ってハッシュ値を求め排他的論理和によるマスクをほどこし、0x00のパディングを付加した結果をRSA暗号化プリミティブで暗号化します。

復号の際はRSA復号化したビット列にたいして最後に付加した0x00が正しく復元されていることを確認します。次にメッセージ部分とパディング部分のビット列からもとのシード部分を復元し、復元したシード値を使ってメッセージ部分を復元します。シード値が正しく復元されていない場合は最終的に復元されるパディング部分に影響するはずで、このようにすることで、復号側でシード値を知らなくてもパディングに単なる固定値を使用するよりはるかに堅牢な暗号化が可能となります。

OAEPや次に説明するPSSで使用するハッシュ関数にはMGF(Mask Generation Function)と呼ばれるハッシュスキームが使用されます。MGFはSHAなど固定サイズのハッシュ関数をベースに所望のサイズのハッシュを得ることができるようにしたハッシュスキームです。



これに対してPSSは署名検証のために開発されたパディングスキームです。PSSでは、適当に選ばれたソルト値を署名のためのハッシュ値に付加した上でそのハッシュ値を求めます。一方、ソルト値にパディングを付加したものに対して、先のハッシュ値とMGFによってマスクし、両者をあわせ固定値0xbcを付加したものを署名値としRSA署名プリミティブを適用します。

検証では、まず固定値0xbcの部分をチェックしRSAプリミティブの処理が正しく行われていることを確認します。その後ハッシュ値をつかってソルト値が復元できるので、署名のときと同じようにメッセージのハッシュ値とソルト値を使ってハッシュ値を求めます。これが、署名によるハッシュ値と一致すれば署名の正当性が検証できたことになります。

このように、PSSではパディングの検証が署名の検証と一体化して行われていることがわかります。

表3-6-2にPKCS#1(RFC8017)で使用されているパディングスキームのオプションをまとめます。

パディング種別	ハッシュ	OID	備考
EMSA-PKCS1-v1_5	MD2	id-md2	limited for compatibility
	MD5	id-md5	limited for compatibility
	SHA-1	id-sha1	limited for compatibility
	SHA-256	id-sha224	
	SHA-256	id-sha256	
	SHA-384	id-sha384	
	SHA-512	id-sha512	
	SHA-512/224	id-sha512-224	
	SHA-512/256	id-sha512-256	
OAEP, PSS	SHA-1	id-sha1	
	SHA-256	id-sha224	
	SHA-256	id-sha256	
	SHA-384	id-sha384	
	SHA-512	id-sha512	
	SHA-512/224	id-sha512-224	
	SHA-512/256	id-sha512-256	

表3-6-2 パディングのハッシュオプション一覧

DHに関する標準

DSAに関する標準

3.7 楕円曲線暗号

「3.4 公開鍵暗号」では整数演算の世界での公開鍵暗号について説明しましたが、楕円曲線上の離散対数演算の一方向性を利用することでも公開鍵暗号を実現できることが知られています。整数演算による公開鍵よりも大幅に短い鍵長でも同等以上の暗号強度が得られるので、攻撃側の計算能力が上がってより強い暗号強度が求められる中で楕円曲線暗号の重要性は増してきています。

[Table 3-7-0: 鍵長と暗号強度](./table3-7-0.md)

楕円曲線暗号は初期には整数演算にくらべると実現がやや複雑になる傾向があって処理速度に課題がありましたが、大幅に短い鍵で同じ暗号強度を得ることができることやより効率的な曲線や実現手法の研究も進み、今日では処理性能の面でも大きく改善しています。

3.7.1 原理

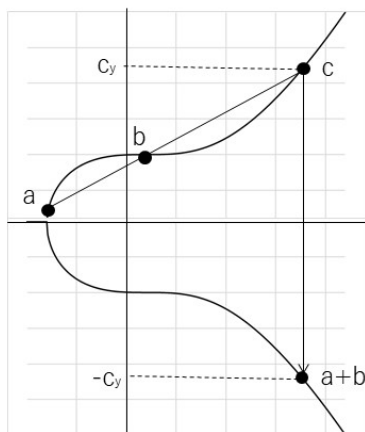
楕円曲線暗号では、まず楕円曲線上の演算を定義します。楕円曲線といっても通常直観的に思い浮かべるような楕円ではなく、次のような数学的に一般化された三次多項式を満たすようなxy座標点の集合を扱います。

$$y^2 = x^3 + ax + b$$

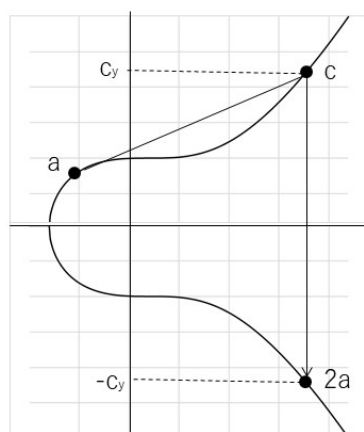
この点の集合は図3-7-1に示すような曲線になります。まず、この曲線の上の点aと点bの加算を、aとbを通る直線が楕円と交わるc点のx軸に対する対称の点 (-yの点) と定義します。

次に、このように定義すると、aの2倍の点は加算においてaとbが同じ値、つまりグラフ上の同じ座標点となった場合に相当します。直観的な意味で曲線がスムーズであれば、曲線上の任意の座標の微分値(傾き)、接線は一つに決まるので、その線の延長上で曲線に交わる点のx軸の対称点をとれば点aを2倍したことになります。従って、これをn回組あわせればaのn倍の座標、つまり任意の座標点のスカラー倍を求めることができることもわかります。

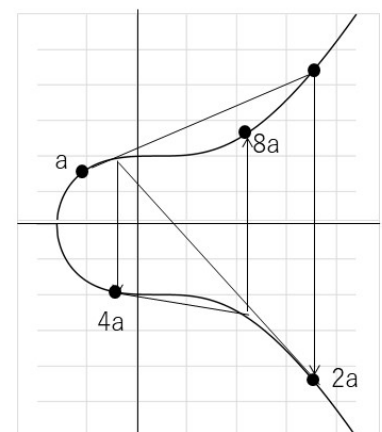
さらに、ax2がそのように求まるなら、aの2のべき乗倍の演算はn回繰り返さなくても図のように求まることになります。これを適当に組み合わせればスカラー倍の演算はいくつかのべき乗演算と加算の組み合わせでより効率的に実現できることもわかります。



点aと点bの加算



点a×2



点a×2^n

楕円曲線上の演算を定義する

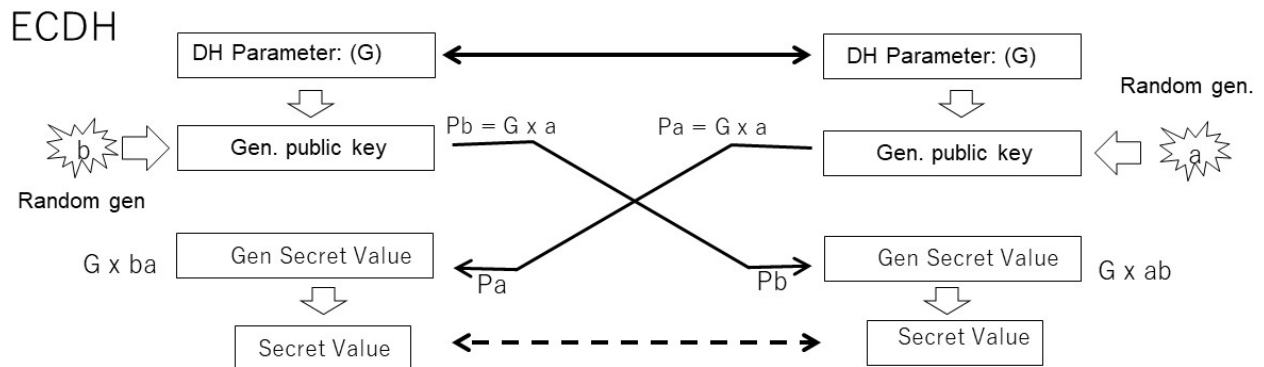
一方、このような楕円曲線上の基になる点(G)、ベースポイントのスカラー倍nの演算は、nが大きな数になると結果の座標xから逆にもとの点を求めることは極めて難しくなる一方向演算であることも知られています。

$$x = nG$$

つまり、スカラー倍演算の一方向性を利用して、係数nを秘密鍵、結果の座標点xを公開鍵とする公開鍵暗号を実現することができ、このような暗号化アルゴリズムを楕円曲線暗号と呼んでいます。

3.7.2 ECDH(楕円曲線デフィーヘルマン)

「3.6.3 ディフィーヘルマン鍵交換」で説明したDHと同じ構造にスカラー倍演算を図3.7.2のようにあてはめて楕円曲線によるディフィーヘルマン鍵交換を実現することができます。演算の原点となる楕円曲線上の座標G（ベースポイント）を共有のパラメータとし、DHの場合と同じようにそれぞれに秘密の乱数値aとbを生成します。それぞれ、Gのa倍、Gのb倍を求めますがこの値からaやbを推定することは極めて難しいので、それを公開鍵として相手に渡すことができます。相手側では、それぞれ受け取った値に自分の秘密鍵aあるいはb倍すると共有の値を得ることができます。



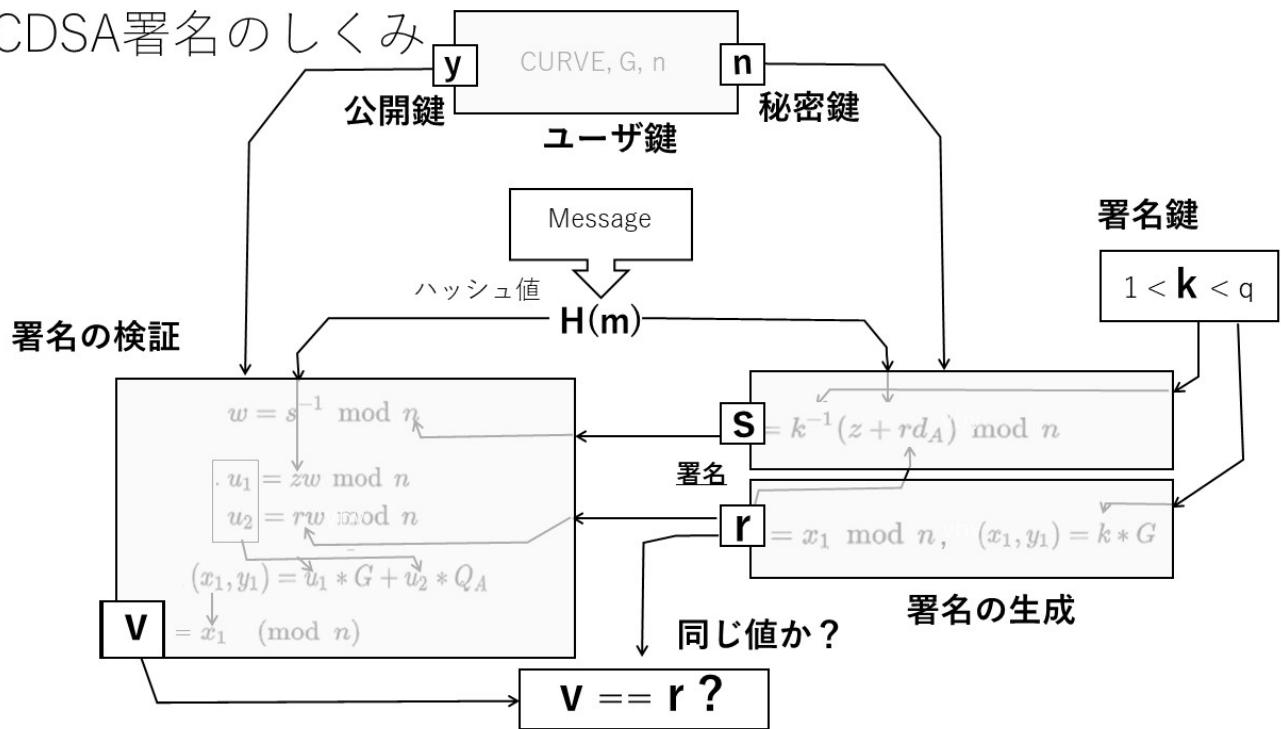
演算内容を見てみると、それぞれの側で行っている演算は演算順序が違うだけなので結果は一致することがわかります。つまり、楕円曲線暗号の世界でもディフィーヘルマン鍵交換が成立しているのです。

abG == baG

3.7.3 ECDSA(楕円曲線デジタル署名)

楕円曲線の演算では巡回群は見つかっていないのでRSAのような手法をとることはできません。しかし、楕円曲線演算の一方方向性を利用してデジタル署名を実現することはできます。図3.7.3に示すようにECDSA(楕円曲線デジタル署名)ではDSAとほぼ同様の構造で署名の生成と検証を行います。

ECDSA署名のしくみ



まず、楕円曲線の定義CURVE、演算のベースとなる曲線上の座標点Gとスカラー係数の最大値nを共通のパラメータとして使用します。

1からn-1の範囲で適当な秘密鍵kを選びます。

署名生成では、署名対象メッセージmのハッシュ値H(m)と秘密鍵kから署名値rとsを求めます。rは署名検証用の値です。

署名検証では、ハッシュ値H(m)と署名値r、sから検証値vを求めます。この値がrと一致していれば正しいメッセージと署名であることが検証できたことになります。メッセージが改ざんされているとH(m)の値が異なり検証値が一致しません。

これらの演算はすべて最初に定めたCURVE曲線上のベースポイントGに対して行われる一方向演算の組み合わせなので、rやsなどから秘密鍵kにたどり着くことはできません。

3.7.4 曲線の種類と標準化

暗号に使用する楕円曲線はあらかじめ標準化しておき利用者が同じ曲線を利用する必要があります。使用する曲線は特異点など脆弱性の原因となるような要素がないことはもちろんですが、曲線の種類によって演算効率が大きく違ってくことも知られています。3次式であらわされる楕円曲線がすべて暗号化アルゴリズムに適しているわけではありません。一般的な楕円曲線のうち特に素体 (Prime Field) と呼ばれる素数の剰余であらわされる曲線と素数2の体 (Binary Field) が楕円暗号のための曲線として深く研究されています。

米国の研究機関NIST (National Institute of Standards and Technology) は早い段階でこうした楕円曲線暗号に使用する曲線の標準化にとりくみ、一連の曲線を推奨曲線として公表しています (SP 800-186: いわゆるNIST曲線)。NISTが標準的に使用を推奨される曲線も素体と素数2の体の中から選ばれています。

一方、国際的活動としてはSECG (Standards for Efficient Cryptography Group) が推奨曲線を発表しており、両者の曲線は対応しているものも多数あります。これらの曲線をベースにIETFではTLSに使用する曲線と利用方法に関してRFC 4492で定義しています。

ECC Brainpool (RFC 5639) : 要加筆

表3-7-4: TLSにおける楕円曲線の標準(RFC 4492)

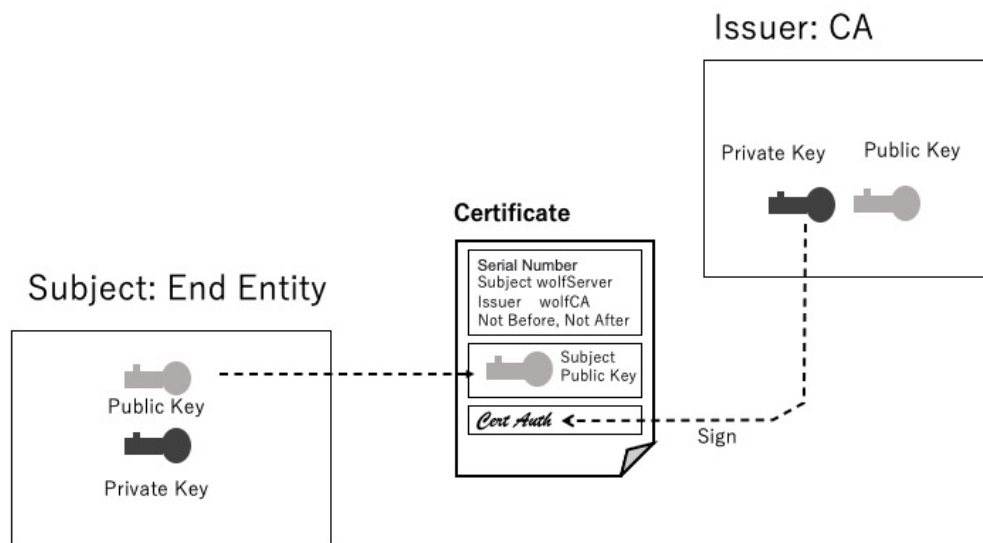
3.7.5 新しい楕円曲線

RFC 7748

3.8 公開鍵証明書

3.8.1 原理

公開鍵証明書は、公開鍵をその所有者の同定、証明書の発行者、署名アルゴリズムなどの属性情報と結びつけるための証明書です。公開鍵は単なるバイナリデータで誰でも完全なコピーを作ってしまうため、公開鍵が所有者本人のものであることを証明するためには別途の方法が必要となります。それが公開鍵証明書です。公開鍵証明書には図3-8-1に示すように鍵の所有者の同定情報その他の属性情報と公開鍵そのもの、またこれら全体に対して署名した公開鍵署名が含まれます。



証明書に含まれている公開鍵は所有者(End Entity)の公開鍵で、対応するプライベート鍵は所有者によって他者から参照されないように適切に管理されます。また、署名は信頼するトレント(通常はCA:Certificate Authority)が自分のプライベート鍵を使って署名します。

このようにして作成された証明書は、CAの公開鍵を使って証明書の署名の正当性を確認することで証明書がそのCAによって署名されたものであること、つまり証明書の真正性(証明書に含まれている公開鍵や所有者の同定情報その他が改竄されたものでないこと)が確認できます。

また、証明書の所有者に対しては、適当なチャレンジに対して所有者のプライベート鍵による署名を求め、その正当性を証明書に含まれている公開鍵でその正当性を確認することによって、真正な所有者であることを確認できます。

証明書の発行、正当性、失効などについては3.9 PKIで説明します。

3.8.2 証明書書式の標準

1) X.509

ITUの定めるX.509は公開鍵証明書の標準規格としてもっとも広く普及、利用されています。X.509は最初のバージョンが1988年に公開、その後改訂されており、IETFではV3が参照されRFC3280として規定されています。

X.509では、証明書の基本的な属性情報として、バージョン、通し番号、アルゴリズムID、発行者情報、証明書の有効期間、主体者(公開鍵の所有者)情報を含みます。また、主体者の公開鍵情報として、公開鍵アルゴリズム、公開鍵自身を含みます。

2) ASN.1

3.9 公開鍵基盤(PKI: Public Key Infrastructure)

3.9.1 PKCS

 Table-3-9-1 表3-1 TLSで参照される主なPKCS

3.9.2 公開鍵による信頼モデル

1) 信頼モデル

認証局同士の信頼関係のモデル。様々な組織が様々な信頼モデルに基づいて認証局を運用しているが、代表的な信頼モデルとして以下のものがある。

単独モデルは1つの認証局が全てのユーザに証明書を発行するモデルであり[6]、企業内の認証局などユーザ数が小規模な場合に用いられる。

階層型モデルは複数の認証局がツリー型の階層構造をなすモデルであり[6]、WebモデルはWebブラウザなどのクライアントアプリケーションに予め認証局の一覧を埋め込むモデルである[6]。

メッシュモデルは階層構造を持たない認証局が相互に認証しあうモデルであり[6]、異なる運用ポリシー(=CPとCPSの記載内容)を持つ「CAドメイン」の間を横断して接続する際に使われる[8]。メッシュモデルでは異なるCAドメインにある認証局同士が相互に認証しあう相互認証でつながりあう[8]。相互認証で用いられる証明書を相互認証証明書という。

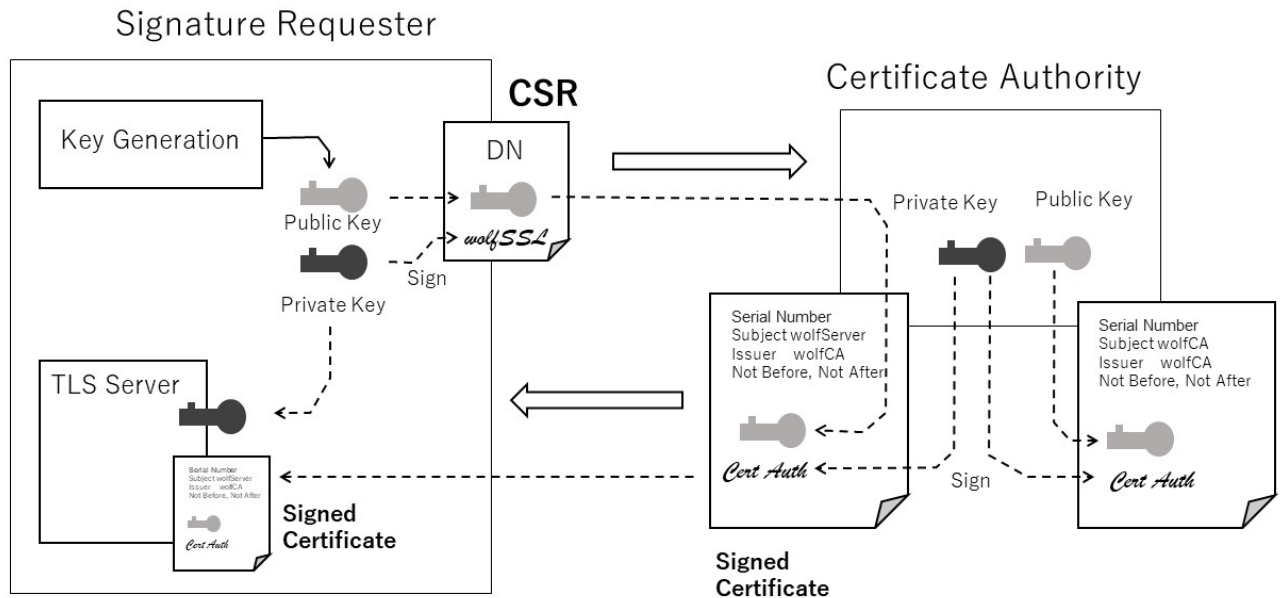
ブリッジCAモデルは認証局同士がブリッジCAという認証局を通して接続するモデルである[6]。メッシュモデルと同じく異なるCAドメイン間を結びつけるときに用いるが、各認証局が相互に認証しあうメッシュモデルと違い、各認証局は1つのブリッジCAとのみ相互認証する。これにより認証局1⇔ブリッジCA⇔認証局2というブリッジCAを媒介とした信頼関係が形成される。認証局の数が多い場合、認証局同士が直接相互認証するメッシュモデルでは相互認証の数が膨大になるが、ブリッジCAモデルではブリッジCAを媒介としたスター型のトポロジーの相互認証が形成されるのみなので、認証パスの数を減らすことができるという利点がある[9]。

3.9.3 証明書のライフサイクル

1) 証明書の発行

CSR(Certificate Signing Request)は、CAによる公開鍵証明書の発行を要求するためのフォーマットです。標準は、当初PKCS#10によって標準化され、RFC2986に引き継がれています。

サーバ認証を受けたいサーバなど公開鍵証明書を必要とする主体(Subject)はCSRによって認証局に対して公開鍵証明書の発行を要求することができます。CSRには主体者の公開鍵とアイデンティティ情報とともに、CSRの偽造を防ぐために主体者の秘密鍵による署名が含まれます。受け取ったCAは、これらの情報に証明書を識別できるシリアル番号、署名者としてのCAのアイデンティティ情報などを付加し、CAの秘密鍵により署名をします。



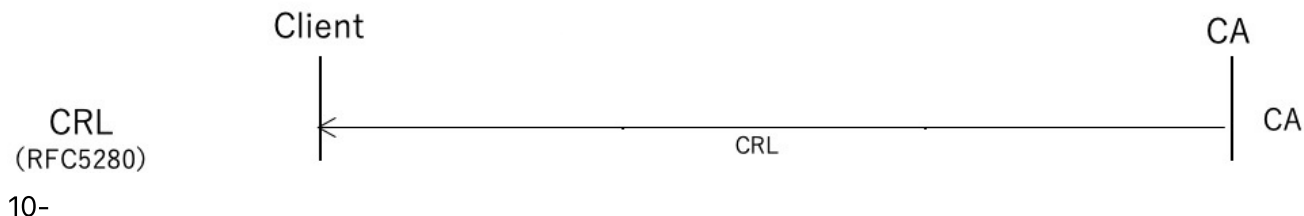
2) 証明書の正当性と失効

公開鍵証明書は、秘密鍵の流出など不測の事態には有効期限内でも失効させることができます。このため、受け取り側は受け取った証明書の有効性について確認する必要があります。証明書の有効性情報の入手は当初CRLやOCSPのようにTLSハンドシェイクのスキープの外で実現されていました。OCSP Stapling ではハンドシェイクの一部としてTLS拡張に取り込まれ、TLS1.3でそれらが整理され現在に至っています。

ここではその経緯を含めてまとめますが、現在ではOCSP Stapling v2以降の使用が推奨されます。また、TLSの基本的なピア認証プロトコルはクライアント、サーバでほぼ対称となっているのですが、クライアント認証におけるOCSPはTLS1.3で初めてサポートされました。

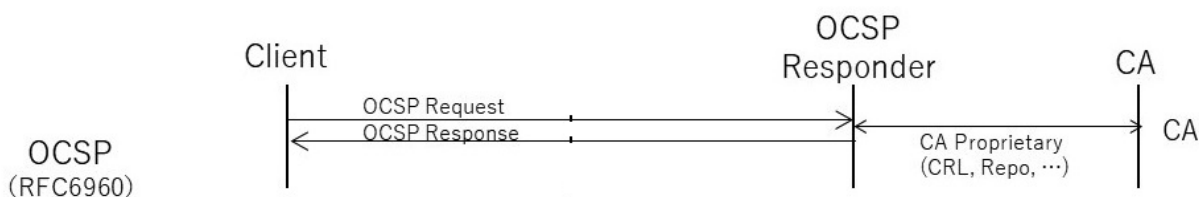
2-1) CRL

初期の証明書有効性の管理メカニズムとしては、失効した証明書の一覧は証明書失効リスト(CRL: Certificate Revocation List) のフォーマットが標準化されました。クライアントはCRLを定期的に入手しておくことで、受け取った証明書の有効性を確認することができます。しかし、クライアント自身がCRL内の証明書情報を確認しなければならず、ネットワークの規模が大きくなりリストのサイズが大きくなるとクライアントにとって負担となってしまいます。



2-2) OCSP

クライアントのそのような負担を軽減するために、受け取った証明書だけに絞ってその有効性をOCSPレスポンスに問い合わせるプロトコルとしてOCSP(RFC6960: Online Certificate Status Protocol)が開発されました。OCSPの場合、クライアントはOCSPレスポンスに対して有効性を確認したい証明書のシリアルナンバーを送り、レスポンスは問い合わせを受けた証明書についての確認結果を返却するのでクライアントの処理負荷が軽減されます。

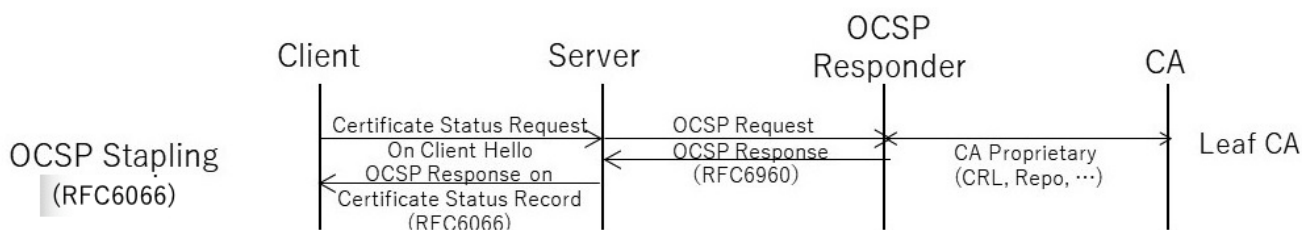


しかし、このネットワーク構成ではレスポンス側にトラフィックが過剰に集中してしまう点が大きな課題となっていました。また、レスポンスがどのような形で失効情報を得るかについては規定されていないので、レスポンスが参照する失効情報自身のリアルタイム性が保証されているわけではありません。

2-3) OCSP Stapling

このように初期のOCSPでは証明書のステータス情報の入手にはTLSとは独立したプロトコルが規定されていました。しかし、その後開発されたOCSP Staplingでは、クライアントはOCSPレスポンスではなくサーバに対してTLSハンドシェイクの一環として証明書の有効性確認要求のプロトコルが標準化されました。これによってクライアントはサーバからの確認結果だけで証明書の有効性を判定できるようになりました。

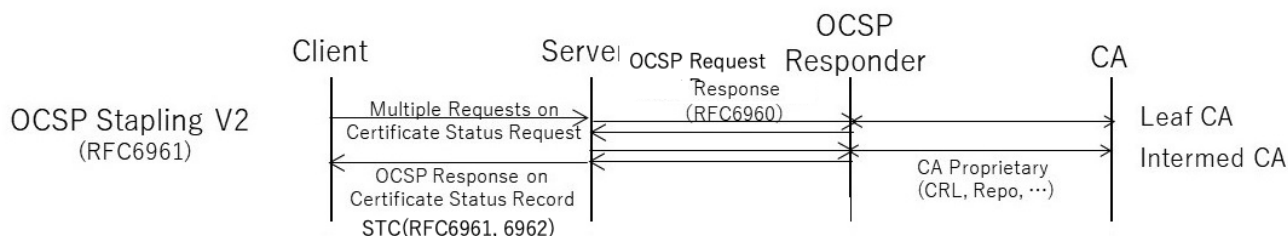
具体的には、クライアントからの要求にはRFC6066でTLS拡張の一つとして追加された証明書ステータス要求 (Certificate Status Request)を使用します。これに伴ってサーバからの応答としてハンドシェイクレコードにCertificateStatusが追加されました。サーバはCertificateStatusレコードにOCSP Responseをのせることによって証明書ステータスを返却します。



2-4) OCSP Stapling Version 2

通常、CAは階層構造を構成していて、それに対応した複数の証明書がチェーンされます。有効性確認は中間CAを含めた証明書についても行う必要がありますが、TLS1.2では一つのClientHelloには一つの証明書ステータス要求拡張しか設けられないという制限があり、中間CA証明書のステータスを含めて要求することができませんでした。

これを解決するためOCSP Stapling バージョン2 として、RFC6961(Multiple Certificate Status Request Extension)で規定が修正拡張されました。バージョン2ではサーバは直接CAに対して有効性確認時のタイムスタンプもクライアントに応答します(RFC6962: Signed Certificate Timestamp)。これによって、クライアントはレスポンスの鮮度を含めて証明書の有効性を確認することができるようになりました。サーバ側も鮮度が許す限りCAへの有効性確認要求を束ねることができるので、問い合わせに対応するCAの負荷を大幅に削減することができるようになりました。



2-5) TLS1.3のOCSP Stapling

TLS1.3では、複数OCSPレスポンドアの証明書ステータスが存在できるようになりこの障害がなくなりました。このためTLS1.3ではクライアントからのステータス確認要求においてはRFC6961の規定する複数証明書ステータス拡張は廃止され、当初のRFC6066の証明書ステータス要求のほうが採用されています。サーバからのレスポンスも Certificate Entry拡張内に対応する証明書とともにRFC6066に準拠したOCSP Responseが設けられました。

TLS1.3ではリクエストとレスポンスのためのTLS拡張が整理されたことにより、サーバからクライアントに対しても同様の証明書ステータス要求を出すこともできるようになりました。この場合、サーバはCertificateRequestに status_reques を乗せて要求を出します(RFC8446 Section 4.4.2.1)。

