Machine Translated by Google

NIST 特別出版物 800-185

SHA-3 派生関数:

cSHAKE、KMAC、TupleHash、ParallelHash

ジョン•ケルシー
シュジェン•チャン
レイ•パールナー

この出版物は、https://doi.org/10.6028/NIST.SP.800-185 から無料で入手できます。

コンピュータセキュリティ

National Institute of Standards and Technology
U.S. Department of Commerce

Machine Translated by Google

NIST 特別出版物 800-185

SHA-3 派生関数:

cSHAKE、KMAC、TupleHash、ParallelHash

ジョン•ケルシー
シュジェン•チャン
レイ•パールナー
コンピュータセキュリティ課
情報基盤研究室

この出版物は、https://doi.org/10.6028/NIST.SP.800-185
から無料で入手できます。

2016年12月

米国商務省ペニー•プリツカー
長官

米国国立標準技術研究所
ウィリー•メイ商務次官（標準技術担当）兼ディレクター

Machine Translated by Google

権限

この出版物は、2014 年の連邦情報セキュリティ近代化法 (FISMA)、44 USC § 3551以降、公法に基づく法的責任に従って NIST によって作成されました。
(PL) 113-283。 NIST は、連邦情報システムの最低要件を含む情報セキュリティ基準およびガイドラインを開発する責任を負っていますが、そのような基準およびガイドラインは、そのようなシステムに対して政策権限を行使する適切な連邦職員の明示的な承認がない限り、国家安全保障システムには適用されません。このガイドラインは、行政管理予算局 (OMB) 回覧 A-130 の要件と一致しています。

この出版物のいかなる内容も、法定権限に基づいて商務長官によって連邦政府機関に義務付けられ拘束力のある基準およびガイドラインに矛盾するとみなされるべきではありません。また、これらのガイドラインは、商務長官、OMB 長官、またはその他の連邦当局者の既存の権限を変更または置き換えるものとして解釈されるべきではありません。この出版物は非政府組織が自発的に使用することができ、米国では著作権の対象ではありません。
ただし、NIST は帰属を歓迎します。

米国国立標準技術研究所特別出版物 800-185
国立研究所立つ。テクノロジー。仕様出版物。 800-185、32 ページ (2016 年 12 月)
コードン: NSPUE2

この出版物は以下から無料で入手できます。
https://doi.org/10.6028/NIST.SP.800-185

実験手順や概念を適切に説明するために、この文書では特定の営利団体、装置、または材料が特定される場合があります。このような特定は、NIST による推奨または承認を意味するものではなく、また、その実体、材料、または機器が必ずしもその目的に利用可能な最良のものであることを意味するものでもありません。

この出版物には、割り当てられた法的責任に従って NIST が現在開発中の他の出版物への参照が含まれている場合があります。この出版物に記載されている情報には、概念や
この方法論は、そのような関連出版物が完成する前であっても、連邦政府機関によって使用される可能性があります。したがって、各出版物が完了するまで、現在の要件、ガイドライン、および手順が存在する場合はそのまま残ります。
工作員。計画と移行の目的で、連邦政府機関は、NIST によるこれらの新しい出版物の開発を綿密に監視することを希望する場合があります。

組織は、パブリックコメント期間中にすべての出版草案をレビューし、NIST にフィードバックを提供することが推奨されます。上記以外の多くの NIST サイバーセキュリティ出版物は、 http://csrc.nist.gov/publications で入手できます。

この出版物に関するコメントは次の宛先に送信できます。

米国国立標準技術研究所
担当 :情報基盤研究所 コンピュータセキュリティ部門
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
電子メール: SP800-185@nist.gov

すべてのコメントは情報公開法 (FOIA) に基づいて公開される対象となります。

Machine Translated by Google

コンピュータシステム技術に関するレポート

米国国立標準技術研究所 (NIST) の情報技術研究所 (ITL) は、国の測定および標準インフラストラクチャに技術的リーダーシップを提供することで、米国の経済と公共の福祉を促進しています。 ITL は、情報技術の開発と生産的な利用を促進するために、テスト、テスト方法、参照データ、概念実証の実装、および技術分析を開発します。 ITL の責任には、連邦情報システムにおける国家安全保障関連情報以外の費用対効果の高いセキュリティとプライバシーのための管理、管理、技術、物理的な標準とガイドラインの開発が含まれます。 Special Publication 800 シリーズは、情報システム セキュリティにおける ITL の研究、ガイドライン、普及活動、および業界、政府、学術団体との協力活動について報告しています。

抽象的な

この勧告では、cSHAKE、KMAC、TupleHash、および ParallelHash の 4 種類の SHA-3 派生関数を指定しており、それぞれ 128 ビットおよび 256 ビットのセキュリティ強度に合わせて定義されています。 cSHAKE は、連邦情報処理標準 (FIPS) 202 で定義されている、SHAKE 関数のカスタマイズ可能なバリアントです。KMAC (KECCAKメッセージ認証コード) は、 KECCAKに基づく可変長メッセージ認証コード アルゴリズムです。擬似ランダム関数としても使用できます。 TupleHash は、入力文字列のタプルをハッシュするように設計された可変長ハッシュ関数です

些細な衝突もなく。 ParallelHash は、非常に長いメッセージを並列でハッシュできる可変長ハッシュ関数です。

キーワード

認証。暗号化。 cシェイク;カスタマイズ可能なSHAKE機能。ハッシュ関数;情報セキュリティー;誠実さ;ケチャック; KMAC;メッセージ認証コード。並列ハッシュ。パラレルハッシュ; PRF;擬似乱数関数; SHA-3;振ってください。タプルのハッシュ化。タプルハッシュ。

謝辞

著者らは、 KECCAKチームのメンバーである Guido Bertoni、Joan Daemen、Michaël Peeters、Gilles Van Assche のフィードバックに感謝します。著者らは、レビューしてくれた同僚にも感謝しています。
この文書の草案を作成し、その開発に貢献します。

Machine Translated by Google

目次

Machine Translated by Google

付録一覧

テーブルのリスト

Machine Translated by Google

1 導入

連邦情報処理標準 (FIPS) 202、 SHA-3 標準: 置換ベースのハッシュおよび拡張可能出力関数[1] では、4 つの固定長ハッシュ関数 (SHA3-224、SHA3-256、SHA3-384、および SHA3- 512)、および 2 つの拡張可能出力関数 (XOF)、

SHAKE128とSHAKE256。これらの SHAKE (セキュア ハッシュ アルゴリズムKECCAK)関数は、新しい種類の暗号化プリミティブです。以前のハッシュ関数とは異なり、期待されるセキュリティ強度に基づいて名前が付けられています。

FIPS 202は、SHA-3 標準の基礎となるアルゴリズム [2] KECCAKから派生したさまざまな関数間のドメイン分離のための柔軟なスキームもサポートしています。ドメインを分離すると、異なる名前付き関数 (SHA3-512 や SHAKE128 など) が無関係になります。 cSHAKE (SHAKE のカスタマイズ可能なバージョン) は、このスキームを拡張して、以下で説明するように、ユーザーが関数の使用をカスタマイズできるようにします。

カスタマイズは、プログラミング言語における強力な型付けに似ています。このようなカスタマイズにより、2 つの異なるカスタマイズ文字列を使用して 1 つの関数を計算する可能性は非常に低くなります。
同じ答えが得られます。したがって、異なるカスタマイズ文字列 (キーのフィンガープリントと電子メールの署名など) を使用した 2 つの cSHAKE 計算には関連性がありません。これらの結果の一方を知っていても、攻撃者にはもう一方に関する情報が与えられません。

この勧告は、2 つの cSHAKE バリアント、cSHAKE128 および cSHAKE256 をセクション 2 で定義しています。これは、FIPS 202 で定義されたKECCAK[c]スポンジ関数 [3]に基づいています。次に、3 つの追加の SHA-3 派生関数が秒単位で定義されています。 4 ～ 6。直接ではなく新しい機能を提供します。
より基本的な機能から利用できます。彼らです：

- KMAC1281および KMAC256 は、可変長出力の擬似乱数関数(PRF)およびキー付きハッシュ関数を提供します。
- TupleHash128 および TupleHash256 は、入力文字列のタプルを明確にハッシュする関数を提供します[2] 。そして
- ParallelHash128 および ParallelHash256。プロセッサの並列処理を利用して長いメッセージをより迅速にハッシュするための効率的なハッシュ関数を提供します。

この勧告で定義されている 4 つの関数 (cSHAKE、KMAC、TupleHash、および
ParallelHash - 次の共通のプロパティがあります。

- これらはすべて、FIPS 202 で指定された関数から派生しています。
- cSHAKE を除くすべての関数は、cSHAKE に関して定義されています。

---

[1] KMAC は、 KECCAKメッセージ認証コードの略です。

[2] TupleHash は 1 つ以上の入力文字列のタプルを処理し、結果のハッシュ値の計算にすべての文字列の内容、文字列の数、各文字列の特定の内容を組み込みます。したがって、何らかの変更 (ある入力文字列から隣接する入力文字列にバイトを移動する、入力タプルから空の文字列を削除するなど) を行うと、異なる結果が生じる可能性が非常に高くなります。

Machine Translated by Google

- すべてはユーザー定義のカスタマイズ文字列をサポートします。
- すべては任意のビット長の可変長出力をサポートします。 KMAC、TupleHash、および ParallelHash には、要求された出力長が変更されると関数が完全に変更されるという追加の特性があります。それ以外の点では入力が同一であっても、要求された出力長が異なる場合にこれらの関数を呼び出すと、一般に関連のない出力が生成されます。

- すべてが 2 つのセキュリティ強度 (128 ビットと 256 ビット) をサポートします。

これらの機能については、以下の特定のセクションで詳しく説明します。さらに、付録 B では、これらの関数を使用して、任意の正の整数Rに対して整数 {0, 1, 2, ..., R－1} にほぼ均一に分布する出力を生成する方法が指定されています。

Machine Translated by Google

2 用語集

この文書では、ビットはCourier Newフォントで示されています。バイトは通常、ASCII 文字 0 ～ 9 および A ～ F の 2 桁の 16 進数として、先頭に接頭辞「0x」を付けて書き込まれます。 2 進数表現では、バイトは下位ビットから最初に書き込まれますが、16 進数表現では、バイトは上位桁から最初に書き込まれます。例: 0x01 = 10000000 および 0x80 = 00000001 これらのビット順序付け規則は、セクション 2 で確立された規則に従います。 FIPS 202 の B.1。このドキュメントでは文字列は二重引用符で囲まれています。

文字列は、長さが 8 ビットの倍数で、0 ビットとそれに続く各連続文字の 7 ビット ASCII 表現で構成されるビット列として解釈されます。

2.1用語と頭字語

少し	2 進数: 0または1。
CMAC	暗号ベースのメッセージ認証コード。
cシェイク	カスタマイズ可能なSHAKE機能。
ドメインの分離	関数の場合、入力が複数のドメインに割り当てられないように、入力を異なるアプリケーション ドメインに分割すること。
拡張可能な出力関数 (XOF)	出力を任意の長さに拡張できるビット文字列の関数。
FIPS	連邦情報処理標準。
ハッシュ関数	出力の長さが固定されているビット文字列の関数。多くの場合、出力は入力の圧縮表現として機能します。
HMAC	キー付きハッシュ メッセージ認証コード。
ケチャック	すべてのスポンジ関数のファミリーは、基礎となる関数としてKECCAK-f順列を使用し、パディング ルールとしてマルチレート パディングを使用します。 KECCAK は元々 [2] で規定され、FIPS 202 で標準化されました。
KMAC	KECCAKメッセージ認証コード。
マック	メッセージ認証コード。
NIST	米国国立標準技術研究所。
PRF	擬似乱数関数を参照してください。

Machine Translated by Google

擬似乱数関数 (PRF)	出力が真にランダムな出力と計算上区別できないように、ランダム シードから出力を生成するために使用できる関数。
レート	スポンジ構造では、処理される入力ビット数 基礎となる関数の呼び出しごとに。
SHA-3	安全なハッシュ アルゴリズム-3。
スポンジ構造	[3] で最初に指定された関数の定義方法は、1) 固定長のビット列に対する基礎となる関数、2) パディング ルール、3) レートです。結果として得られる関数の入力と出力はどちらもビット文字列であり、長さは任意です。
スポンジ機能	スポンジの構造に従って定義された関数。固定出力長に特化している可能性があります。
弦	ビットのシーケンス。
XOF	「拡張可能出力関数」を参照してください。

2.2基本操作

$\lceil x \rceil$	実数x の場合、 $\lceil x \rceil$ は厳密にはxより小さくない最小の整数です。たとえば、 $\lceil 3.2 \rceil = 4$、 $\lceil -3.2 \rceil = -3$、および $\lceil 6 \rceil = 6$ です。
0^s	正の整数s の場合、 0^s はs 個の連続する 0 ビットで構成される文字列です。
a mod b	整数aとbのモジュロ演算。 「a mod b」は、 aをbで割った余りを返します。
enc8(i)	0 ～ 255 の整数iの場合、 enc8(i)はiのバイト エンコーディングです。 ビット 0 はバイトの下位ビットです。
レン(X)	ビット文字列X の場合、len(X) はXのビット長です。
X || Y	文字列XとY の場合、 X || YはXとYを連結したものです。たとえば、次のようになります。 11001 || 010 = 11001010。

2.3その他の内部関数

このセクションでは、SHA-3 派生関数の定義で使用される文字列エンコード、パディング、および部分文字列関数について説明します。

Machine Translated by Google

2.3.1 整数からバイト文字列へのエンコーディング

2 つの内部関数left_encodeとright_encode は、整数をバイト文字列としてエンコードするために定義されています。どちらの関数も、整数を非常に大きな最大値 22040－1 までエンコードできます。

left_encode(x) は、 xのバイト文字列表現の前にバイト文字列の長さを挿入することで、文字列の先頭から明確に解析できる方法で、整数 x をバイト文字列としてエンコードします。

right_encode(x) は、 xのバイト文字列表現の後にバイト文字列の長さを挿入することで、文字列の末尾から明確に解析できる方法で、整数 x をバイト文字列としてエンコードします。

関数enc8()を使用して個々のバイトをエンコードすると、これら 2 つの関数は次のように定義されます。

right_encode(x):
有効条件: $0 \leq x < 22040$

1. nを$28n > x$となる最小の正の整数とします。
2. x1、 x2、…、 xn を、 $x = \Sigma\ 28(ni)\ xi$ (i = 1 ～ nの場合)を満たすxの
   Base256 エンコードであるとします。
3. Oi = enc8(xi)、 i = 1 ～ nとします。
4. On+1 = enc8(n) とします。
5. O = O1 ||を返します。 O2 || … ||オン||オン+1。

left_encode(x):
有効条件: $0 \leq x < 22040$

1. nを$28n > x$となる最小の正の整数とします。
2. x1、 x2、 …、 xnを、 $x = \Sigma\ 28(ni)\ xi$ (i = 1 ～ nの場合)を満たすxの Base256
   エンコードとする。
3. Oi = enc8(xi)、 i = 1 ～ nとします。
4. O0 = enc8(n) とします。
5. O = O0 ||を返します。 O1 || … ||オンー1 ||の上。

例として、right_encode(0) は 00000000 10000000 を生成し、left_encode(0) は 10000000 00000000 を生成します。

2.3.2 文字列エンコーディング

encode_string 関数は、文字列S の先頭から明確に解析できる方法でビット文字列をエンコードするために使用されます。この関数は次のように定義されます。

エンコード文字列(S):
有効条件: $0 \leq len(S) < 22040$

1. left_encode(len(S)) を返します|| S.

Machine Translated by Google

例として、encode_string(S) (Sが空の文字列 "") は 10000000 00000000 となります。

ビット文字列Sがバイト指向でない場合 (つまり、len(S)が 8 の倍数でない場合)、encode_string(S) から返されるビット文字列もバイト指向ではないことに注意してください。ただし、len(S)が 8 の倍数の場合、encode_string(S) の出力の長さも 8 の倍数になります。

2.3.3 パディング

bytepad(X, w) 関数は、整数wのエンコードを入力文字列 X の先頭に付加し、バイト長がwの倍数になるバイト文字列になるまで結果をゼロで埋めます。一般に、bytepad はエンコードされた文字列で使用することを目的としています。バイト文字列 bytepad(encode_string(S), w) は先頭から明確に解析できますが、bytepad はすべての入力文字列に対して明確なパディングを提供しません。

bytepad() の定義は次のとおりです。

バイトパッド(X, w):
有効条件: w > 0

1. z = left_encode(w) || X. 2.
while len(z) mod 8 ≠ 0: z || 0
z = 3.
while (len(z)/8) mod w ≠ 0: z = z ||
00000000 4. z を返し
ます。

2.3.4 部分文字列

パラメータaとbを、ビット文字列X内の特定の位置を示す非負の整数とします。
非公式には、substring(X, a, b) 関数は、ビット位置a、 a+1、...、 b－1の値を含むビット文字列Xから部分文字列を返します。より正確には、部分文字列関数は以下に定義されているように動作します。入力文字列と出力文字列のすべてのビット位置には、ゼロからインデックスが付けられることに注意してください。
したがって、文字列の最初のビットは位置 0 にあり、 n ビット文字列の最後のビットは位置n-1 にあります。

部分文字列(X, a, b):

1. a ≥ bまたはa ≥ len(X) の場
合:空の文字列を返します。
2. b ≤ len(X) の場合、位置
aから位置b－1までのXのビットを返します。
3. それ以外の場合:
Xの位置aから位置len(X)－1までのビットを返します。

Machine Translated by Google

3 シーシェイク

3.1概要

cSHAKE の 2 つのバリアント (cSHAKE128 と cSHAKE256) は、FIPS 202 で指定されている SHAKE 関数とKECCAK[c] 関数に関して定義されています。 cSHAKE128 は 128 ビットのセキュリティ強度を提供し、cSHAKE256 は 256 ビットのセキュリティ強度を提供します。

3.2パラメータ

どちらの cSHAKE 関数も 4 つのパラメータを取ります。

- Xはメイン入力ビット文字列です。ゼロを含む任意の長さ 3 にすることができます。
- Lは、要求された出力長 4 をビット単位で表す整数です。
- Nは関数名のビット文字列で、NIST が cSHAKE に基づいて関数を定義するために使用します。
  cSHAKE 以外の機能が必要ない場合、 Nには空の文字列が設定されます。
- Sはカスタマイズビット文字列です。ユーザーはこの文字列を選択して、
  関数。カスタマイズが必要ない場合、 S は空の string5 に設定されます。

cSHAKE の実装は、合理的に入力文字列と出力長のみをサポートする場合があります。
それはバイト全体です。その場合、小数バイトの入力文字列または 8 の倍数ではない出力長の要求はエラーになります。

NとSが両方とも空の文字列の場合、cSHAKE(X, L, N, S) は、FIPS 202 で定義されている SHAKE と同等です。

cSHAKE128(X, L, "", "") = SHAKE128(X, L) そして
cSHAKE256(X, L, "", "") = SHAKE256(X, L)。

cSHAKE は、任意の 2 つのインスタンスに対して次のように設計されています。

cSHAKE(X1、 L1、 N1、 S1)および
cSHAKE(X1、 L1、 N2、 S2)、

---

[3] この文書で文字列が「任意の長さ」であると指定されている場合、理論上の制限 (例: 22040-1ビット) が適用される場合があります。
この制限は、整数エンコード スキーム left_encode および right_encode によって課されます。 2.3.1.
この制限を超えると、文字列をエンコードできなくなります。このドキュメントの残りの部分では、 22040-1ビットのような非常に大きな制限は、制限がまったくない状態で同じ意味で扱われることがよくあります。

[4] 要求された出力長がゼロ、つまり L=0 の場合、 cSHAKE、KMAC、TupleHash、および ParallelHash は出力として空の文字列を返します。

[5] パラメーターのデフォルト値をサポートするコンピューティング言語では、この関数を実装する自然な方法は、 NとSのデフォルト値を空の文字列に設定することです。

Machine Translated by Google

N1 = N2 およびS1 = S2でない限り、2 つのインスタンスは無関係な出力を生成します。これには以下が含まれることに注意してください
N1 とS1 が空の文字列の場合。つまり、カスタマイズを含む cSHAKE は、FIPS 202 で指定されている通常の SHAKE 関数からドメイン分離されています。

3.3定義

cSHAKE は、SHAKE またはKECCAK[c]に関して次のように定義されます。SHAKE への呼び出しの結果を返すか (NとSが両方とも空の文字列の場合)、 KECCAK(c)への呼び出しの結果を返します。
NとSのパディングされたエンコーディングが入力文字列Xに連結されています。

cSHAKE128(X、 L、N、 S):
有効条件: len(N)< 22040およびlen(S)< 22040

1. N = "" およびS = "" の場合:
SHAKE128(X, L)を返します;
2. それ以外の場合:
KECCAK[256](bytepad(encode_string(N) || encode_string(S), 168) || X || 00, L)を返します。

cSHAKE256(X、 L、N、 S):
有効条件: len(N)< 22040およびlen(S)< 22040

1. N = "" およびS = "" の場合:
SHAKE256(X, L)を返します;
2. それ以外の場合:
KECCAK[512](bytepad(encode_string(N) || encode_string(S), 136) || X || 00, L)を返します。

数値 168 と 136 は、 KECCAK[256]とKECCAK[512]のレート(バイト単位)であることに注意してください。
それぞれスポンジ機能。これらの定義のCourier Newフォントの文字 00 は、 2 つのゼロ ビットを指定します。

3.4関数名入力の使用方法

cSHAKE 関数には、関数名(N) を提供するために使用できる入力文字列が含まれています。
これは、NIST が SHA-3 派生関数を定義する際に使用することを目的としており、NIST6 で定義された値にのみ設定する必要があります。このパラメータは、関数名によるドメイン分離のレベルを提供します。 cSHAKE のユーザーは、自分自身の名前を作成しないでください。そのようなカスタマイズが、カスタマイズ文字列S の目的です。これについては、セクション 2 で説明します。 3.5. Nの非標準値は、将来の NIST 定義関数との相互運用性の問題を引き起こす可能性があります。

[6] NIST は常に関数名N をバイト指向の値にします。

Machine Translated by Google

3.5カスタマイズ文字列の使用

cSHAKE 関数には、ユーザーが関数の使用をカスタマイズできるようにする入力文字列(S)も含まれています。たとえば、cSHAKE128 を使用してキー フィンガープリント (公開キーのハッシュ値) を計算する場合は、次のように使用します。

cSHAKE128(public_key, 256, "", "キーの指紋"),

ここで、「キーの指紋」はカスタマイズ文字列Sです。

後で、同じユーザーが、署名用に別の cSHAKE 計算をカスタマイズすることを決定する可能性があります。
Eメール：

cSHAKE128(email_contents, 256, "", "電子メール署名"),

ここで、「電子メール署名」はカスタマイズ文字列Sです。

カスタマイズ文字列は、これら 2 つの cSHAKE 値間の衝突を回避することを目的としています。攻撃者が何らかの方法で 1 つの計算 (電子メールの署名) を強制的に実行することは非常に困難です。
異なるS値が使用された場合、他の計算 (キー フィンガープリント) と同じ結果が得られます。

カスタマイズ文字列の長さは22040 未満にすることができます。ただし、実装によっては、受け入れられるSの長さが制限される場合があります。

Machine Translated by Google

4KMAC

4.1概要

KECCAKメッセージ認証コード(KMAC) アルゴリズムは、 KECCAKに基づく PRF およびキー付きハッシュ関数です。これは可変長の出力を提供しますが、SHAKE や cSHAKE とは異なり、要求された出力の長さを変更すると、関連性のない新しい出力が生成されます。 KMAC には、それぞれ cSHAKE128 と cSHAKE256 から構築された 2 つのバリアント、KMAC128 と KMAC256 があります。 2 つの亜種は、技術的なセキュリティ特性において多少異なります。それにもかかわらず、ほとんどのアプリケーションでは、セクション 2 で説明するように、十分な長さのキーが使用されていれば、どちらのバリアントも最大 256 ビットのセキュリティ強度をサポートできます。 8.4.1.

4.2パラメータ

どちらの KMAC 関数も次のパラメータを取ります。

- K は、ゼロを含む任意の長さのキー ビット文字列 7 です。
- Xはメイン入力ビット文字列です。ゼロを含む任意の長さにすることができます。
- Lは、要求された出力長をビット単位で表す整数です。
- Sは、ゼロを含む任意の長さのオプションのカスタマイズ ビット文字列です。カスタマイズが必要ない場合、 S は空の文字列に設定されます。

4.3定義

KMAC は、キーKのパディングされたバージョンを入力Xおよび要求された出力長 L のエンコーディングと連結します。次に、結果は、要求された出力長L、名前N ="KMAC" = 11010010 10110010 10000010とともに cSHAKE に渡されます。 110000109 、およびオプションのカスタマイズ文字列S。

KMAC128(K、 X、 L、 S):
有効条件: len(K) < 22040および0 ≤ L < 22040およびlen(S) < 22040

1. newX = bytepad(encode_string(K), 168) || × || right_encode(L)。
2. cSHAKE128(newX, L, “KMAC”, S) を返します。

---

[7] KMAC の承認された使用では、セクション 1 で説明するように、 Kの長さが少なくとも必要なセキュリティ強度である必要があります。 8.4.1.

[8] セクション 2 で説明したように、関数が XOF として使用されない限り、この関数からの出力には22040-1ビットの制限があることに注意してください。 4.3.1.

[9] バイナリ表現では、セクション 2 で指定されているように、この文書ではバイトが下位ビットから最初に書き込まれることに注意してください。 2.

Machine Translated by Google

KMAC256(K、 X、 L、 S):
有効条件: len(K) <22040および0 ≤ L < 22040およびlen(S) < 22040

1. newX = bytepad(encode_string(K), 136) || × || right_encode(L)。
2. cSHAKE256(newX, L, “KMAC”, S) を返します。

数値 168 と 136 は、 KECCAK[256]とKECCAK[512]のレート(バイト単位)であることに注意してください。
それぞれスポンジ機能が備わっています。

4.3.1 任意長出力の KMAC

KMAC の一部のアプリケーションでは、出力の生成が開始されるまで必要な出力ビット数がわからない場合があります。これらのアプリケーションでは、KMAC を XOF として使用することもできます (つまり、出力を任意の長さに拡張できます)。これは cSHAKE の動作を模倣します。

XOF として使用される場合、KMACXOF128(K, X, L, S) および KMACXOF256(K, X, L , S) のステップ 1 の right_encode(0) に示すように、エンコードされた出力の長さを 0 に設定することによって KMAC が計算されます。) 以下の疑似コード。概念的には、XOF モードの KMAC は無限長の出力文字列を生成し、呼び出し元は必要なだけ出力文字列のビットを使用するだけです。 XOF モードでの KMAC の切り捨てられた出力は、次の疑似コードで与えられる関数 KMACXOF128(K, X, L, S) または KMACXOF256(K, X, L, S) によって計算できます。

KMACXOF128(K、 X、 L、 S):
有効条件: len(K) < 22040および0 ≤ Lおよびlen(S) < 22040

1. newX = bytepad(encode_string(K), 168) || × || right_encode(0)。
2. cSHAKE128(newX, L, “KMAC”, S) を返します。

KMACXOF256(K、 X、 L、 S):
有効条件: len(K) <22040、0 ≤ L 、およびlen(S) < 22040

1. newX = bytepad(encode_string(K), 136) || × || right_encode(0)。
2. cSHAKE256(newX, L, “KMAC”, S) を返します。

Machine Translated by Google

5 タプルハッシュ

5.1概要

TupleHash は、可変長出力を備えた SHA-3 派生のハッシュ関数で、入力文字列のタプル (一部またはすべてが空の文字列である可能性があります) を明確な方法で単純にハッシュするように設計されています。このようなタプルは、ゼロを含む任意の数の文字列で構成され、この例では ("a"、"b"、"c"、...、"z") のように括弧で囲まれた一連の文字列または変数として表されます。書類。

TupleHash は、たとえば、タプル ("abc" ,"d") に対して計算された TupleHash が TupleHash とは異なるハッシュ値を生成するように、ハッシュ用の文字列のシーケンスを組み合わせる汎用的で誤用防止の方法を提供するように設計されています。タプル ("ab","cd") で計算されますが、
残りの入力パラメータはすべて同じに保たれ、文字列エンコードを行わない場合、結果として得られる 2 つの連結文字列は同一です。

TupleHash は、128 ビットと 256 ビットの 2 つのセキュリティ強度をサポートします。要求された出力の長さを含め、関数への入力を変更すると、ほぼ確実に最終出力が変更されます。

5.2パラメータ

TupleHash は次のパラメータを取ります。

- Xは 0 個以上のビット文字列のタプルであり、その一部またはすべてが空の文字列である場合があります。
- Lは、要求された出力長をビット単位で表す整数です。
- Sは、ゼロを含む任意の長さのオプションのカスタマイズ ビット文字列です。カスタマイズが必要ない場合、 S は空の文字列に設定されます。

5.3定義

TupleHash は、入力文字列のシーケンスを明確な方法でエンコードし、文字列の末尾で要求された出力長をエンコードし、その結果を関数名 (N) の「TupleHash」 = 00101010 10101110 00001110 00110110 10100110 とともに cSHAKE に渡します。 00010010 10000110 11001110 00010110、およびオプションのカスタマイズ文字列S。

Xがn 個のビット文字列のタプルの場合、 X[i]を 0 から番号付けする i 番目のビット文字列とします。 TupleHash 関数は、疑似コードで次のように定義されます。

タプルハッシュ128(X, L, S):
有効条件: 0 ≤ L < 22040およびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 ～ n の場合:
   z = z || encode_string(X[i])。
4. newX = z || right_encode(L)。

Machine Translated by Google

5. cSHAKE128(newX, L, “TupleHash”, S) を返します。

タプルハッシュ256(X, L, S):
有効条件: 0 ≤ L < 22040およびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 〜 n の場合:
   z = z || encode_string(X[i])。
4. newX = z || right_encode(L)。
5. cSHAKE256(newX, L, “TupleHash”, S) を返します。

5.3.1 任意長出力のタプルハッシュ

TupleHash の一部のアプリケーションでは、出力の生成が開始されるまで必要な出力ビット数がわからない場合があります。これらのアプリケーションでは、TupleHash を XOF として使用することもできます (つまり、出力を任意の長さに拡張できます)。これは cSHAKE の動作を模倣します。

XOF として使用される場合、TupleHash は、以下のTupleHashXOF128(X, L, S) および TupleHashXOF256(X, L, S) 疑似コードのステップ 1 の right_encode(0) に示すように、エンコードされた出力の長さを 0 に設定することによって計算されます。
概念的には、XOF モードの TupleHash
無限長の出力文字列を生成し、呼び出し元は必要なだけ出力文字列のビットを使用するだけです。 XOF モードでの
TupleHash の切り捨てられた出力は、次の疑似コードで与えられる関数 TupleHashXOF128(X, L, S) または
TupleHashXOF256(X, L, S) によって計算できます。

タプルハッシュXOF128(X, L, S):
有効条件: 0 ≤ Lおよびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 〜 n の場合:
   z = z || encode_string(X[i])。
4. newX = z || right_encode(0)。
5. cSHAKE128(newX, L, “TupleHash”, S) を返します。

タプルハッシュXOF256(X, L, S):
有効条件: 0 ≤ Lおよびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 〜 n の場合:
   z = z || encode_string(X[i])。
4. newX = 5. z || right_encode(0)。
cSHAKE256(newX, L, “TupleHash”, S) を返します。

Machine Translated by Google

6 パラレルハッシュ

6.1概要

ParallelHash10 の目的は、最新のプロセッサで利用可能な並列処理を利用して、非常に長い文字列の効率的なハッシュ化をサポートすることです。 ParallelHash は、128 ビットおよび 256 ビットのセキュリティ強度をサポートし、可変長出力も提供します。入力パラメーターを ParallelHash に変更すると、要求された出力長であっても、関連性のない出力が生成されます。このドキュメントで定義されている他の関数と同様に、ParallelHash はユーザーが選択したカスタマイズ文字列もサポートしています。

6.2パラメータ

ParallelHash は次のパラメータを受け取ります。

- Xはメイン入力ビット文字列です。ゼロを含む任意の長さ11にすることができます。
- Bは、並列ハッシュのブロック サイズ (バイト単位) です。 0 < B <のような任意の整数を指定できます。
  22040。
- Lは、要求された出力長をビット単位で表す整数です。
- Sは、ゼロを含む任意の長さのオプションのカスタマイズ ビット文字列です。カスタマイズが必要ない場合、 S は空の文字列に設定されます。

6.3定義

ParallelHash は、入力ビット文字列X を、それぞれ長さBバイトの連続した重複しないブロックのシーケンスに分割し、各ブロックのハッシュ値を個別に計算します。
最後に、これらのハッシュ値が結合され、「ParallelHash」の関数名(N) = 00001010 10000110 01001110 10000110 00110110 00110110 10100110 00110110 00010010 10000110 11 とともに cSHAKE に渡されます。 001110 00010110、オプションのカスタマイズ文字列 S、およびいくつかのエンコードされた整数値 (以下の疑似コードに示されています)、関数の最終ハッシュ値を生成します。

ParallelHash 関数は、疑似コードで次のように定義されます。

ParallelHash128(X、 B、 L、 S):
有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ L < 22040およびlen(S) < 22040

1. n = (len(X)/8) / B 。

[10] 将来的には、他の NIST 承認ハッシュ関数用の汎用並列ハッシュ モードが開発される可能性があります。ここでの関数 (つまり、ParallelHash) は、特に cSHAKE に基づいており、したがってKECCAK に基づいています。

[11] ここで、 len(X)/B < 22040 、 B はセクション 2 で定義されているバイト単位のブロック サイズです。 6.2.脚注 2 で指定されているように、NIST は、このような不条理に大きな制限を、まったく制限がないことと交換可能なものとして扱います。

Machine Translated by Google

2. z = left_encode(B)。
3. i = 0 ～ n－1 の場合:
z = z || cSHAKE128(部分文字列(X, i*B*8, (i+1)*B*8), 256, "", "")。
4.z = z || right_encode(n) || right_encode(L)。
5. newX = 6. z.
cSHAKE128(newX, L, “ParallelHash”, S) を返します。

ParallelHash256(X、 B、 L、 S):
有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ L < 22040およびlen(S) < 22040

1. n = (len(X)/8) / B 。
2. z = left_encode(B)。
3. i = 0 ～ n－1 の場合:
z = z || cSHAKE256(部分文字列(X, i*B*8, (i+1)*B*8), 512, "", "")。
4.z = z || right_encode(n) || right_encode(L)。
5.newX = z。
6. cSHAKE256(newX, L, “ParallelHash”, S) を返します。

6.3.1 任意の長さの出力を持つ ParallelHash

ParallelHash のアプリケーションによっては、出力の生成が開始されるまで必要な出力ビット数がわからない場合があります。これらのアプリケーションでは、ParallelHash を XOF として使用することもできます (つまり、出力を任意の長さに拡張できます)。これは cSHAKE の動作を模倣します。

XOF として使用される場合、ParallelHashXOF128(X, B, L, S) および ParallelHashXOF256(X, B, L , S)のステップ 1 の right_encode(0) に示すように、ParallelHash はエンコードされた出力の長さを 0 に設定することによって計算されます。)
以下の疑似コード。概念的には、XOF モードの ParallelHash
無限長の出力文字列を生成し、呼び出し元は必要なだけ出力文字列のビットを使用するだけです。 XOF モードでの
ParallelHash の切り捨てられた出力は、次の疑似コードで与えられる関数 ParallelHashXOF128(X, B, L, S) または
ParallelHashXOF256(X, B, L, S) によって計算できます。

ParallelHashXOF128(X、 B、 L、 S):
有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ Lおよびlen(S) < 22040

1. n = (len(X)/8) / B 。
2. z = left_encode(B)。
3. i = 0 ～ n－1 の場合:
z = z || cSHAKE128(部分文字列(X, i*B*8, (i+1)*B*8), 256, "", "")。
4.z = z || right_encode(n) || right_encode(0)。
5.newX = z。
6. cSHAKE128(newX, L, “ParallelHash”, S) を返します。

ParallelHashXOF256(X、 B、 L、 S):

Machine Translated by Google

有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ Lおよびlen(S) < 22040

1. n = (len(X)/8) / B 。
2. z = 左エンコード(B)。
3. i = 0 ～ n－1 の場合:
z = z || cSHAKE256(部分文字列(X, i*B*8, (i+1)*B*8), 512, "", "")。
4. z = z || right_encode(n) || right_encode(0)。
5. newX = 6. z.

cSHAKE256(newX, L, “ParallelHash”, S) を返します。

Machine Translated by Google

7 実装に関する考慮事項

7.1事前計算

cSHAKE は、関数名Nとカスタマイズ文字列Sに対応するための基礎となるKECCAK-f関数 [1]へのすべての呼び出しがrビットの整数倍を処理するように定義されます (rはレート パラメーター)。実装では、このパディングされたNとSのブロックを cSHAKE で処理した結果を事前計算できるため、複数の cSHAKE 実行で同じNとSの選択を再利用する場合でもパフォーマンスが低下することはありません。 TupleHash と ParallelHash は cSHAKE に関して定義されているため、これと同じ事前計算をこれらの関数の実装でも利用できます。

KMAC は、 NとSの処理結果、およびキーKの処理結果を事前計算できます。
したがって、固定の事前計算されたカスタマイズ文字列とキーを使用する KMAC128 は、SHAKE128 と同じくらい効率的に入力文字列を処理します。

7.2限定的な実装

cSHAKE、KMAC、TupleHash、および ParallelHash 関数は、広範囲の可能な入力 (不当に長い入力や小数バイトを含む入力を含む) を受け入れ、広範囲の可能な出力長を生成するように定義されています。ただし、特定の実装では、処理できる入力と、生成される許容される出力の長さを制限することが許容されます。

たとえば、これらの関数の実装を、65536 バイト以下の出力を生成するように制限したり、全バイトの出力のみを生成したり、バイト文字列のみ (小数バイトは禁止) を入力として受け入れるように制限したりすることは許容されます。さらに、特定の限定された用途のみを目的とした実装では、処理する入力のセットがさらに制限される場合があります。

たとえば、文字列の 6 タプルを処理するためだけに使用され、常に「アドレス タプル」のカスタマイズ文字列を使用する TupleHash256 の実装は許容されます。

これらの関数のいずれかの実装に、処理できない入力セットが与えられる可能性がある場合、その実装はエラー状態を通知し、出力の生成を拒否します。

7.3 ParallelHash での並列処理の活用

ParallelHash の特定の実装では、その実装を許可された値の小さなサブセットに制限することが許可されています。たとえば、特定の実装が、同様にBを同じ値に制限する別の実装と相互運用することのみが期待される場合、B の単一の値のみを許可することは許容されます。

ParallelHash は、逐次処理のみが利用可能な場合でも、簡単かつ合理的に効率的な方法で実装できます。ただし、メッセージの個々のブロックを並行して処理できる場合は、はるかに高速な実装が可能です。ブロックサイズの選択

この場合、 B はParallelHash の効率に大きな影響を与える可能性があります。 ParallelHash は、並列処理を適用できるマシンであれば、原則としてその並列処理の恩恵を受けることができるように設計されています。たとえば、4 つのブロックを並行してハッシュできるマシンと、

Machine Translated by Google

32 ブロックを並列にハッシュすると、利用可能なすべての並列処理能力の恩恵を受けることができます。

Machine Translated by Google

8 セキュリティに関する考慮事項

8.1主張されるセキュリティ強度

cSHAKE、KMAC、TupleHash、ParallelHash はすべて、128 ビットと 256 ビットという 2 つのセキュリティ強度を主張するために定義されています。

cSHAKE128、KMAC128、TupleHash128、および ParallelHash128 はそれぞれ 128 ビットのセキュリティ強度を提供します。これは、特定の出力長 L について、同じ出力長のハッシュ関数には存在しない、 2128未満の作業を必要とするこれらの関数の 1 つに対する一般的な攻撃は存在しないことを意味します。同様に、cSHAKE256、KMAC256、TupleHash256、および ParallelHash256 はそれぞれ、一般的な攻撃に対して 256 ビットのセキュリティ強度を提供します。

8.2関数名とカスタマイズ文字列のセキュリティ プロパティ

8.2.1 法的なNおよびSに対する SHAKE と同等のセキュリティ

関数名Nとカスタマイズ文字列Sを選択した場合、cSHAKE128(X, L, N, S) は SHAKE128(X, L) とまったく同じセキュリティ プロパティを持ちます。 cSHAKE256(X, L, N, S) には、SHAKE256(X, L)とまったく同じセキュリティ プロパティがあります。 NまたはSには「弱い」値はありません。

8.2.2 異なるNとSによる無関係な出力

(n1, s1) と(n2, s2) が 2 つの名前とカスタマイズ文字列のペアであり、 n1 ≠ n2 またはs1 ≠ s2であるとします。さらに、 q1 とq2 が要求された出力の長さであるとします。すると、cSHAKE(X, q1, n1, s1) と cSHAKE(X, q2, n2, s2) はXの無関係な関数として扱うことができます。
つまり、出力長がそれぞれq1 とq2 である 2 つの完全に異なる関数であるかのように扱うことができます。これは、たとえば次のことを意味します。

- キーk1 とk2。k1 = cSHAKE(x1, q1, n1, s1)、 k2 = cSHAKE(x2, q2, n2, s2)、両方のキーは秘密ではあるが関連する数量x1 とx2 から導出されます。関連するキー攻撃の影響を受けません (cSHAKE 関数の主張されているセキュリティ強度よりも複雑ではありません)。

- cSHAKE128(x1, L, n1, s1) = cSHAKE128(x2, L, n2, s2)となるような衝突を見つけるには、min(2L/2, 2128)程度の計算量が必要になります。同様に、cSHAKE256(x1, L, n1, s1) = cSHAKE256(x2, L, n2, s2)のような衝突を見つけるには、min(2L/2, 2256)程度の計算量が必要になります。

KMAC、TupleHash、および ParallelHash は cSHAKE から派生しているため、これらのプロパティを継承します。具体的には：

- これらの各関数は、他の関数の出力とは無関係な出力を生成します。
  たとえば、KMAC の出力 (入力の任意のセットに対する) と TupleHash (入力の任意のセットに対する) の間には関係がありません。 • これらの関数のいずれについても、別のカスタマイズ文字列を使用すると、無関係な出力が得られます。
  したがって、 s1 ≠ s2 の場合、ParallelHash(X, B, L, s1) と ParallelHash(X, B, L, s2) には特別な関係がないことが期待されます。

Machine Translated by Google

XOF モードで使用する場合を除き、KMAC、TupleHash、および ParallelHash には追加の機能があります。
同じカスタマイズ文字列、ブロックサイズ、キーなどを使用しても、
出力の長さが異なる関数は、残りの関数とは無関係な関数として扱うことができます。
入力。したがって、たとえば、ParallelHash(X, B, q1, S) および ParallelHash(X, B, q2, S) を扱うことができます。
入力Xと出力長q1 およびq2 をそれぞれ持つ独立したハッシュ関数として。

cSHAKE はこのプロパティを共有しないことに注意してください。 q1< q2の場合、cSHAKE(X, q1, N, S) は次のプレフィックスになります。
cSHAKE(X, q2, N, S)。これは、cSHAKE が SHAKE および他の XOF と共有するプロパティです。
このプロパティについては、FIPS 202 の付録 A.2 で詳しく説明されています。

8.3衝突とプリイメージ

これらの関数はすべて、可変出力長をサポートしています。攻撃者がこれらの関数の衝突やプリイメージを見つける難易度は、要求されるセキュリティ強度と出力長の両方によって異なります。

128 ビットのセキュリティ強度が謳われている cSHAKE128 のような関数は、出力の長さに関係なく、 2128ワークによる衝突またはプリイメージ攻撃に対して脆弱になる可能性があります。一般に、出力が長くても、これらの攻撃に対するセキュリティは向上しません。ただし、出力が短いと、関数がこれらの攻撃に対して脆弱になる可能性があります。 Lビットの出力では、衝突攻撃には min(2L/2, 2128)の作業が必要となり、プリイメージ攻撃には少なくとも min(2L, 2128)の作業が必要になります。

8.4 KMAC を安全に使用するためのガイダンス

柔軟性と有用性を最大限に高めるために、KMAC 関数は任意のサイズの出力長とキー長 (最大22040-1ビット) に対して定義されています。ただし、そのような出力とキーの長さのすべてが一致するわけではありません。
安全です。

8.4.1 KMAC キーの長さ

入力キーの長さは、セキュリティの強度に最も直接的に変換されるパラメータです。少数の既知の (MAC、平文) ペアが与えられた場合、攻撃者はキーKを見つけるために最大2len(K)の操作を必要とします。

この勧告の適用では、必要なセキュリティ強度よりも短い長さの入力キーKを選択してはならない。暗号化アルゴリズムと鍵サイズの選択に関するガイダンスは、[4] で入手できます。

8.4.2 KMAC 出力長

出力の長さは、KMAC のもう 1 つの重要なセキュリティ パラメータです。これにより、オンライン推測攻撃が MAC タグの偽造に成功する確率が決まります。特に、攻撃者は、偽造が成功するたびに、平均して2L の無効な (メッセージ、MAC) ペアを送信する必要があります。 L はオンライン攻撃にのみ影響するため、メッセージ認証に KMAC を使用するシステムは、特定のキーで許可される検証失敗の総数を制限することで、短いLを悪用する攻撃を軽減できます。

Machine Translated by Google

MAC として使用する場合、この勧告のアプリケーションは出力長Lを選択してはならない
これは 32 ビット未満であり、慎重なリスク分析が実行された後にのみ 64 ビット未満の出力長を選択する必要があります。

特定のパラメータ設定に対する KMAC のセキュリティ プロパティを説明するために、表 1 に、他のいくつかの承認された MAC アルゴリズムと、KMAC の同等の設定を例として示します。リストされているアルゴリズムでは、関連するタグ (つまり、AES-CMAC の 128 ビット タグ、HMAC-SHA256 の 256 ビット タグ、HMAC-SHA512 の 512 ビット タグ) の切り捨てが想定されていないことに注意してください。異なる MAC アルゴリズムの同等の設定では、同じ出力は得られません。

表 1: KMAC と以前に標準化された MAC アルゴリズムの同等のセキュリティ設定

既存の MAC アルゴリズム	KMAC相当品
AES-CMAC (K、テキスト)	KMAC128 (K、テキスト、 128、 S)
HMAC-SHA256 (K、テキスト)	KMAC256 (K、テキスト、 256、 S)
HMAC-SHA512 (K、テキスト)	KMAC256 (K、テキスト、 512、 S)

Machine Translated by Google

KECCAK[c]に関する KMAC、TupleHash、および ParallelHash

FIPS 202 はKECCAK[c]関数を指定しており、これに基づいて SHA-3 関数と SHAKE 関数が構築されています。 KMAC、TupleHash、および ParallelHash は、セクション 2 で指定されているように、cSHAKE に関して定義されています。 3. この付録では、KMAC、TupleHash、ParallelHash、および XOF モードのこれらの関数は、KECCAK[c]に関して直接定義されます。これらの定義は、Secs の cSHAKE に関して行われた定義とまったく同じです。 4、5、6。

KMAC128(K、 X、 L、 S):
有効条件: len(K) < 22040および0 ≤ L < 22040およびlen(S) < 22040

1. newX = bytepad(encode_string(K), 168) || X || right_encode(L)。
2. T = bytepad(encode_string(“KMAC”) || encode_string(S), 168)。
3. KECCAK[256](T || newX || 00, L) を返します。

KMAC256(K、 X、 L、 S):
有効条件: len(K) < 22040および0 ≤ L < 22040およびlen(S) < 22040

1. newX = bytepad(encode_string(K), 136) || X || right_encode(L)。
2. T = bytepad(encode_string(“KMAC”) || encode_string(S), 136)。
3. KECCAK[512](T || newX || 00, L) を返します。

KMACXOF128(K、 X、 L、 S):
有効条件: len(K) < 22040および0 ≤ Lおよびlen(S) < 22040

1. newX = bytepad(encode_string(K), 168) || X || right_encode(0)。
2. T = bytepad(encode_string(“KMAC”) || encode_string(S), 168)。
3. KECCAK[256](T || newX || 00, L) を返します。

KMACXOF256(K、 X、 L、 S):
有効条件: len(K) <22040、0 ≤ L 、およびlen(S) < 22040

1. newX = bytepad(encode_string(K), 136) || X || right_encode(0)。
2. T = bytepad(encode_string(“KMAC”) || encode_string(S), 136)。
3. KECCAK[512](T || newX || 00, L) を返します。

タプルハッシュ128(X, L, S):
有効条件: 0 ≤ L < 22040およびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 ～ n の場合:
   z = z || encode_string(X[i])。
4. newX = z || right_encode(L)。
5. T = bytepad(encode_string(“TupleHash”) || encode_string(S), 168)。
6. KECCAK[256](T || newX || 00, L) を返します。

Machine Translated by Google

タプルハッシュ256(X, L, S):
有効条件: 0 ≤ L < 22040およびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 ～ n の場合:
   z = z || encode_string(X[i])。
4. newX = z || right_encode(L)。
5. T = bytepad(encode_string(“TupleHash”) || encode_string(S), 136)。
6. KECCAK[512](T || newX || 00, L) を返します。

タプルハッシュXOF128(X, L, S):
有効条件: 0 ≤ Lおよびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 ～ n の場合:
   z = z || encode_string(X[i])。
4. newX = z || right_encode(0)。
5. T = bytepad(encode_string(“TupleHash”) || encode_string(S), 168)。
6. KECCAK[256](T || newX || 00, L) を返します。

タプルハッシュXOF256(X, L, S):
有効条件: 0 ≤ Lおよびlen(S) < 22040

1. z = ""。
2. n = タプルX内の入力文字列の数。
3. i = 1 ～ n の場合:
   z = z || encode_string(X[i])。
4. newX = z || right_encode(0)。
5. T = bytepad(encode_string(“TupleHash”) || encode_string(S), 136)。
6. KECCAK[512](T || newX || 00, L) を返します。

ParallelHash128(X、 B、 L、 S):
有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ L < 22040およびlen(S) < 22040

1. n = (len(X)/8) / B 。
2. z = 左エンコード(B)。
3. i = 0 ～ n－1 の場合:
   z = z || KECCAK[256](substring(X, i*B*8, (i+1)*B*8) || 1111, 256)。
4. z = z || right_encode(n) || right_encode(L)。
5. newX = 6. z.
T = bytepad(encode_string(“ParallelHash”) || encode_string(S), 168)。
7. KECCAK[256](T || newX || 00, L) を返します。

ParallelHash256(X、 B、 L、 S):

Machine Translated by Google

有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ L < 22040およびlen(S) < 22040

1. n = (len(X)/8) / B 。
2. z = left_encode(B)。
3. i = 0 ～ n－1 の場合:
z = z || KECCAK[512](substring(X, i*B*8, (i+1)*B*8) || 1111, 512)。
4. z = z || right_encode(n) || right_encode(L)。
5. newX = 6. z.
T = bytepad(encode_string(“ParallelHash”) || encode_string(S), 136)。
7. KECCAK[512](T || newX || 00, L) を返します。

ParallelHashXOF128(X、 B、 L、 S):
有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ LおよびLen(S) < 22040

1. n = (len(X)/8) / B 。
2. z = 左エンコード(B)。
3. i = 0 ～ n－1 の場合:
z = z || KECCAK[256](substring(X, i*B*8, (i+1)*B*8) || 1111, 256)。
4.z _ = z || right_encode(n) || right_encode(0)。
5. newX = 6. z.
T = bytepad(encode_string(“ParallelHash”) || encode_string(S), 168)。
7. KECCAK[256](T || newX || 00, L) を返します。

ParallelHashXOF256(X、 B、 L、 S):
有効条件: 0 < B < 22040および len(X)/B < 22040および
0 ≤ LおよびLen(S) < 22040

1. n = (len(X)/8) / B 。
2. z = left_encode(B)。
3. i = 0 ～ n－1 の場合:
z = z || KECCAK[512](substring(X, i*B*8, (i+1)*B*8) || 1111, 512)。
4.z = z || right_encode(n) || right_encode(0)。
5.newX = z 。 _
6. T = bytepad(encode_string(“ParallelHash”) || encode_string(S), 136)。
7. KECCAK[512](T || newX || 00, L) を返します。

Machine Translated by Google

付録 B - 範囲へのハッシュ (参考情報)

XOF、PRF、cSHAKE、KMAC、TupleHash、ParallelHash などの可変長出力のハッシュ関数を使用すると、 $0 \leq X < R$の範囲内の整数X (このドキュメントでは0..R－1と表記)を生成することが簡単にできます。上記の関数が一様確率変数に近似していると仮定すると、次の方法では、その範囲にわたって一様分布に非常に近い出力が生成されます。

0..R－1の範囲の整数にハッシュするには、次の手順を実行します。

1. k = lg(R) + 128 とします。
2. 少なくともkビットの長さを要求されたハッシュ関数を呼び出します。結果のビット列を次のようにします。
   Z.
3. N = bits_to_integer(Z) mod R とします。ここで、 bits_to_integer関数は以下で定義されます。

このプロセスの最後では、変数Nには、次の値に非常に近い整数が含まれます。
0..R－1の範囲に均一に分布します。 $0 \leq t < R$のような任意の出力値tについて、
次のステートメントは true12 です。

|確率(N=t) – 1/R| ≤ 2－128/R。

言い換えれば、このプロセスの出力には非常に小さなバイアスが含まれます。あまり価値はないだろう
整数があった場合よりも、このプロセスの結果として現れる可能性が多かれ少なかれあります。
0 から R-1 までの整数から均一にランダムに選択されます。

この手法は、SHAKE、cSHAKE、KMAC、TupleHash、または ParallelHash に適用できます。
結果として許容される限り、特定の範囲内の整数が必要な場合はいつでも
整数 {0, 1,..., の一様分布からのこの非常に小さな偏差を持つ整数
R－1}。

bits_to_integer関数は、次のようにビット文字列を整数に変換します。

bits_to_integer (b1, b2,…, bn):

1. (b1, b2,…, bn)をビット列の最上位から最下位までのビットとします。
   ビット。

2. = Σ 2 =1 (に)び。

3. (x) を返します。

---

[12] 実際には、境界はこれよりわずかにタイトです。 w = ビット列Zのビット長(w ≥ lg(R) + 128)の場合、 |Prob(N=t) – 1/R|となります。 ≤ 2-w。

Machine Translated by Google

付録 C—参考資料

[1] 米国国立標準技術研究所、 SHA-3 標準: 順列ベースのハッシュおよび拡張可能出力関数、連邦情報処理標準 (FIPS) 出版物 202、2015 年 8 月、37 ページhttp://dx.doi.org/10.6028/ NIST.FIPS.202。

[2] G. Bertoni、J. Daemen、M. Peeters、および G. Van Assche、 KECCAKリファレンス、バージョン 3.0、2011年 1 月 14 日、69 ページhttp://keccak.noekeon.org/Keccak-reference-3.0.pdf [2016 年 12 月 21 日にアクセス]。

[3] G. Bertoni、J. Daemen、M. Peeters、および G. Van Assche、暗号化スポンジ関数、バージョン 0.1、2011年 1 月 14 日、93 ページhttp://sponge.noekeon.org/CSF-0.1.pdf [2016 年 12 月 21 日にアクセス]。

[4] E. Barker、鍵管理に関する推奨事項、パート 1: 一般、 NIST Special Publication (SP) 800-57 Part 1 Revision 4、国立標準技術研究所、2016 年 1 月、160 ページhttp://dx.doi.org /10.6028/ NIST.SP.800-57pt1r4。