IETF 119 120 における Security AREA と関連技術開発の動向レポート

2024年08月18日

塚本　明

目次

- セキュリティーのニーズと技術全体の相互関係

① SBOM （Software Bill of Materials）
- SBOM 情報義務化の背景
- SBOM の 2 つのレベルについて
- SBOM ファイル生成方法と開発現場の実情

② IETF
- セキュリティ強化の動きと連動した組織改編
- IETF における Security AREA 内の活動

③ セキュアブート
- 要求される暗号関連のプログラミング技術
- セキュアブートの実装例
- ハッシュ関数と暗号アルゴリズムの選択

④ CI/CD
- SBOM 生成と署名の自動化

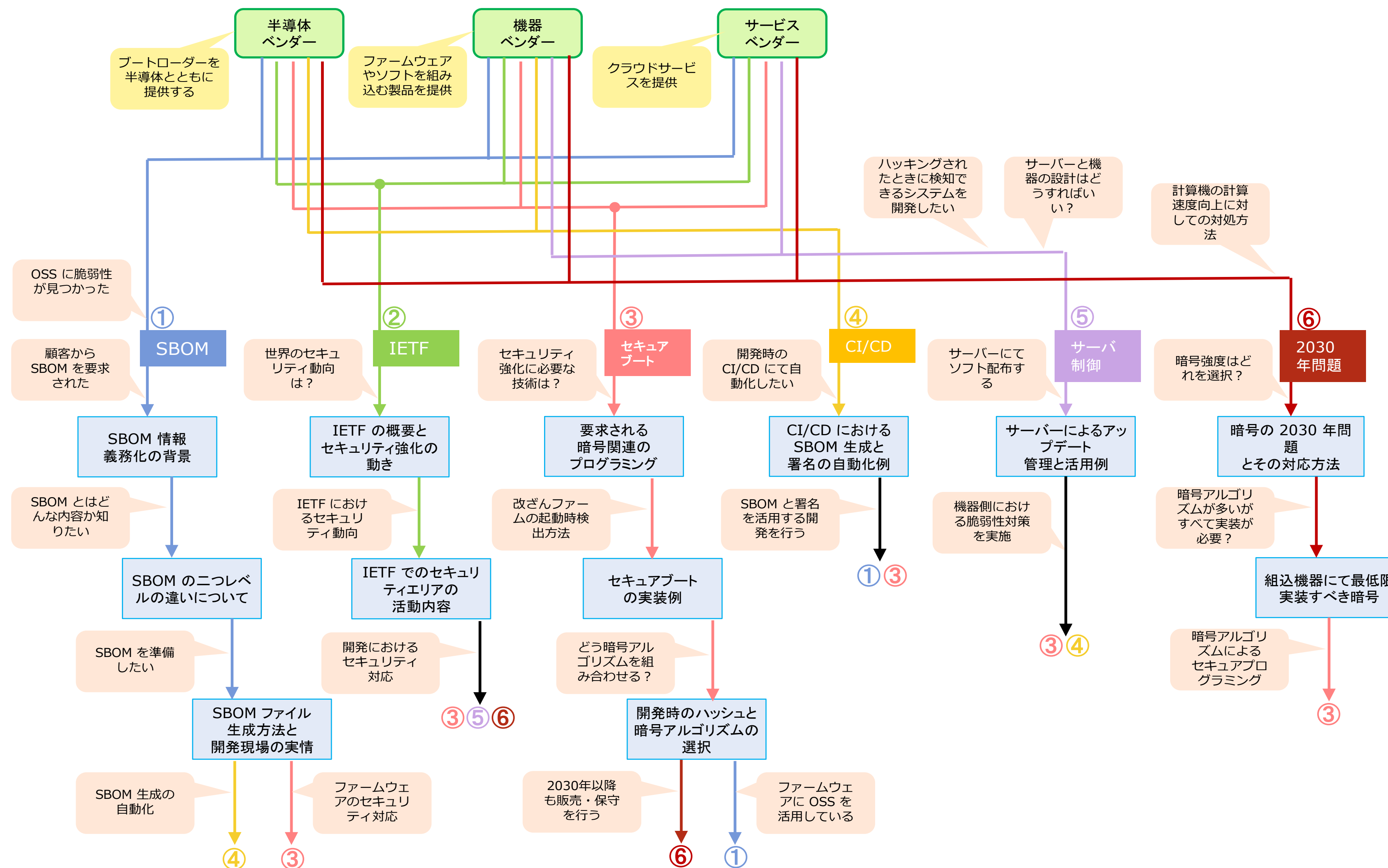
⑤ サーバーによるアップデートの管理

⑥ 暗号の 2030 年問題
- 対応方法
- 組込機器にて最低限実装すべき暗号

- 付録
  - 耐量子暗号アルゴリズム対応に向けて

技術全体の背景と相互関係 (1/2)

- セキュリティーニーズと、本資料の各章との対応



技術全体の背景と相互関係 (2/2)

- 各キーワード（緑色）は製品・サービスのセキュリティーとして相互補完の関係にある
- SBOM（Software Bill of Materials）とセキュリティーの関係
  - アプリ・ファームウェア・ブートローダーには、内部に多くのオープンソースパッケージが使われている
  - 機器ベンダーは、あるパッケージの特定バージョンに脆弱性の発見情報が公開された場合、脆弱性を修正したパッケージに入れ替えて、アップデートしたファームウェアを提供したい
  - 購入した製品のファームに脆弱性のあるパッケージが活用されていた場合は、可及的に早く修正したファームに更新したい
  - ソフトウェアパッケージのリストである SBOM（Software Bill of Materials）にてパッケージ名とバージョンを管理することで、パッケージの脆弱性のあるバージョンを特定ができるようになる
- セキュアブート(Secure Boot) と SBOM の関係
  - 上記により脆弱性を修正したファームウェアを適応する場合、対象機器がすでにハッキングされていた場合はセキュリティーを担保できない
  - そのため機器が起動時にファームが改ざんされていないことを検証しながら起動する技術をセキュアブートという
  - セキュアブートの実現には、ハッシュ関数・暗号化アルゴリズムの技術が必要
- サーバーによる管理による脆弱性更新との関係
  - オンラインにてアップデートできることが理想
  - この時に、遠隔サーバーから対象機器がハッキングされないことを検証（リモートアテステーション）でき、機器のファームのバージョンを把握して適切なアップデート（ライフサイクルマネージメント）ができる必要がある
  - このリモートアテステーションとライフサイクルマネージメントをハードのサポートを得て実現する技術が Trusted Execution Environment Provisioning (TEEP)である
- 暗号の203年問題
  - 暗号アルゴリズムは年々暗号強度の強いアルゴリズムに移行する必要があり、製品での対策方法 z
- 現在上記の緑のキーワードの技術は IETF、Linux Foundation 等にて技術開発と標準化が進行中

SBOM 情報義務化の背景（1/2）

- 背景・目的
  - 企業や米国政府が調達する製品・サービスにセキュリティ脆弱性が見つかった時に敏速に対応ができるよう、内部に使用されているソフトウェアコンポーネント（SBOM）の情報が必要になった
  - 製品・サービスを構成するフトウェアに含まれるオープンソースパッケージの飛躍的な増大
    - 1990年代は100以下、現在は2000以上の場合あり
      78% のビジネスがオープンソースを活用して構築 https://jobera.com/open-source-software-statistics/
      98% のアプリがオープンソースを使用 https://zipdo.co/open-source-software-statistics/
- SBOM （全部読むのは大変なので次頁以降に要約、要求内容が少ないものから多いものまで）
  - NIST (National Institute of Standards and Technology)
    - 米国政府が定める調達ガイドライン（大統領令 14028）
    - https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-security-supply-chains-software-1
  - NTIA（National Telecommunications and Information Administration）
    - 米国の通信事業者が求める調達要件についての指針
    - https://www.ntia.gov/page/software-bill-materials
  - DoD（米国国防総省の調達要件）
    - https://www.congress.gov/bill/117th-congress/senate-bill/4543/text
  - EU Cyber Resilience Act
    - CE マークを満たすための基準
    - https://www.cisa.gov/sites/default/files/2023-09/EU%20Commission%20SBOM%20Work_508c.pdf

SBOM 情報義務化の背景（2/2）

- 前項からいくつか抜粋
  - 国防総省の調達要件部分（対象とするソフトの範囲が少ない）
    - 製品に使用されているオープンソースのソフトウェアに対して SBOM を要求

SEC. 1627. REQUIREMENT FOR SOFTWARE BILL OF MATERIALS.

(a) Requirement for Software Bill of Materials.--

(1) In general.--The Secretary of Defense shall amend the Department of Defense Supplement to the Federal Acquisition Regulation to require a software bill of materials (SBOM) for all noncommercial software created for or acquired by the Department of Defense.

  - EU Cyber Resilience Act の SBOM 記述
    - SBOM 作成を要求するが、必ずしも SBOM の公開を要求していない

- **Manufacturers to draw up an SBOM** in a commonly used format covering at the very least the top-level dependencies of the product
- **No requirement** to make the SBOM publicly available
- SBOM to be included in the **technical documentation** and, upon request, to be provided to **market surveillance authorities**

SBOM に求められるレベル（1/2）

- SBOM 要求にはレベルが 2 つある

レベル 1. 製品・サービスごとの内部で使われているソフトパッケージの SBOM 提供の義務化
レベル 2. 製品・サービスには複数の業者や複数の国をまたがって開発されたソフトウェアコンポーネントが使われることが一般的なため、そのトラッキング情報（いわゆるサプライチェーン情報）が正しいかどうかの検証

レベル 1. 製品ベンダーに SBOM 提供を義務化（IETF は SPDX, CycloneDX の成果を使う立場）

- 製品・サービスには複数のソフトウェアパッケージが使われている。
  - 当初 SBOM に含まれる情報は (1)パッケージ名, (2)バージョン番号, (3)ライセンス種別のうち最初の 2 つ、あるいは 3 つの情報であった。
  - その後、提供者・取得先（Supplier）、依存関係（Dependency）が追加。
  - それらに加えて SBOM 作成者（製品・サービスベンダー）と作成日時。
    - 代表的はファイルフォーマット: SPDX, CycloneDX, SWID
- この時、製品・サービスの購入者が一番提供してほしい情報はオープンソースを使っているときのパッケージ名とバージョン番号である。
- オープンソースの場合は、脆弱性が見つかった時に世の中でお影響が大きいことから、脆弱性を持つパッケージ名とバージョンの情報が公開されることが多い。この情報をもとに脆弱性を持つバージョンを避けて、該当パッケージに対して脆弱性が修正された新しいバージョンにするか、脆弱性が存在しない古いバージョンに戻すことで対処ができるようになる。

過去の例において数年経つと政府調達に従い、一般企業の顧客も同様の SBOM を要求するため準備が必要

SBOM に求められるレベル（2/2)

レベル 2. サプライチェーンにおける SBOM 情報を検証する仕組み（IETF が議論の中心）

- 前項の SBOM 情報の扱いにおいて、次の二つの懸念点を対処するための仕組みが求めらるようになった。

(1) ベンダーから提供された SBOM 情報の真正性を検証できる機能（改ざん等の恐れがあるため）

(2) 場合によっては競合他社に SBOM 情報を知られたくない場合等のため、一部の SBOM 情報を開示せずとも信頼性と追跡性を提供する機能

- 本機能を提供するシステムの技術開発と標準化活動は、IETF の次の WG が中心
  - Supply Chain Integrity, Transparency, and Trust (SCITT)
  - Secure Patterns for Internet CrEdentials (SPICE)

デジタルクレデンシャルにおけるリンク不可(unlinkability)、選択的開示 (selective disclosure)の技術を持たせることにより実現を目指している

- 現在 IETF にて技術開発と標準化の真っ最中 -> IETF の章にて解説

機器ベンダーの対応の必要性は、まだ先になると思われる。先に検証機関の設立が先

SBOM ファイル生成例（1/5）

- SBOM ファイルに含まれる情報の一般的な項目
  - 記述フォーマットは json , xml, yaml どれかとなっているが、json が一般的
  - 当初は項目にある Dependency は一般的でなく、その代わりにライセンス情報があった
  - 次項以降の例では Dependency を取得していない

MINIMUM ELEMENT	SPDX 2.3	CYCLONEDX 1.5
Supplier Name	PackageSupplier	Supplier
Component Name	PackageName	Name
Version of the Component	PackageVersion	Version
Other Unique Identifiers	DocumentNamespace, SPDXID	Purl, cpe, swid
Dependency Relationship	Relationship	Dependencies
Author of SBOM Data	Creator	Author
Timestamp	Created	Timestamp

SBOM ファイル生成例（2/5）

- github をソースコード管理サーバーに使っており、node.js, rubyプログラムの場合
  - 「Export SBOM 」にて出力されるのは SPDX フォーマット

出力例

削除

SBOM ファイル生成例（3/5）

- Yocto をビルドしている場合
  - SBOM を自動生成する方法説明のリンク。出力されるのは SPDX フォーマット
    - Yocto のページ
      - https://docs.yoctoproject.org/dev/dev-manual/sbom.html
      - https://fosdem.org/2024/schedule/event/fosdem-2024-3318-spdx-in-the-yocto-project/
  - “conf/local.conf” に次の行を追加してビルド
    - + INHERIT += “create-spdx“ # spdx ファイル出力を指定
    - + SPDX_PRETTY = “1” # これをつけないと読みにくい json ファイルになる
  - “tmp/deploy/images/MACHINE/” の下に “IMAGE-MACHINE.spdx.json” が生成される
    - 一つにまとまった“IMAGE-MACHINE.spdx.json”ファイルを構成する各ファイルは “tmp/deploy/spdx/MACHINE” にある
    - インストールされているパッケージリスト
      ${DEPLOY_DIR}/images/${MACHINE}/${TARGET_IMAGE_NAME}-${MACHINE}.manifest
    - 各パッケージのパッケージ名、バージョン、ライセンスのリスト
      ${DEPLOY_DIR}/images/${MACHINE}/${TARGET_PACKAGE_NAME}.spdx.json
  - ビルド時にソースの CVE データベースと照合するは次の行を追加
    - INHERIT += "cve-check“
  - “build/tmp/deploy/cve” の下に各パッケージが CVE 対処のパッチが当たっているかのリストが生成
    - 出力例

削除

SBOM ファイル生成例（4/5）

- Python プロジェクトの場合、SBOM4Python と CycloneDX Python が多く使われる
  - CycloneDX Python の例

- requirements.txt があるプロジェクト

```
$ git clone https://github.com/videvelopers/Vulnerable-Flask-App
$ cyclonedx-py environment --of json -o sbom.json
```

削除

- Ripfile があるプロジェクト

```
$ git clone https://github.com/docusign/code-examples-python
$ cyclonedx-py pipenv --of json -o d-sbom.json
```

削除

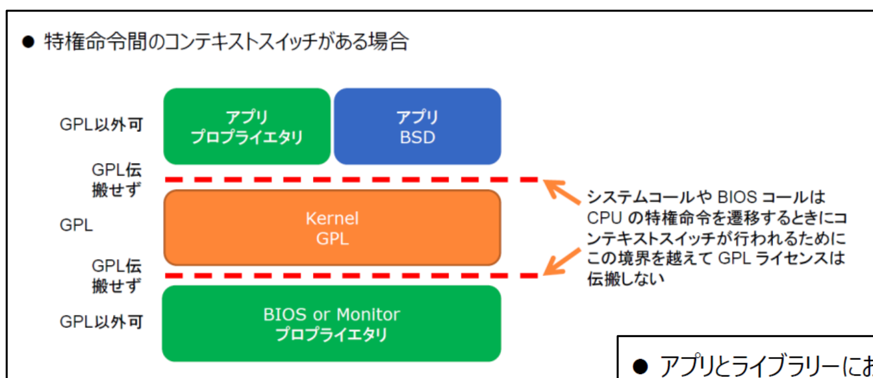
SBOM ファイル生成例（5/5）

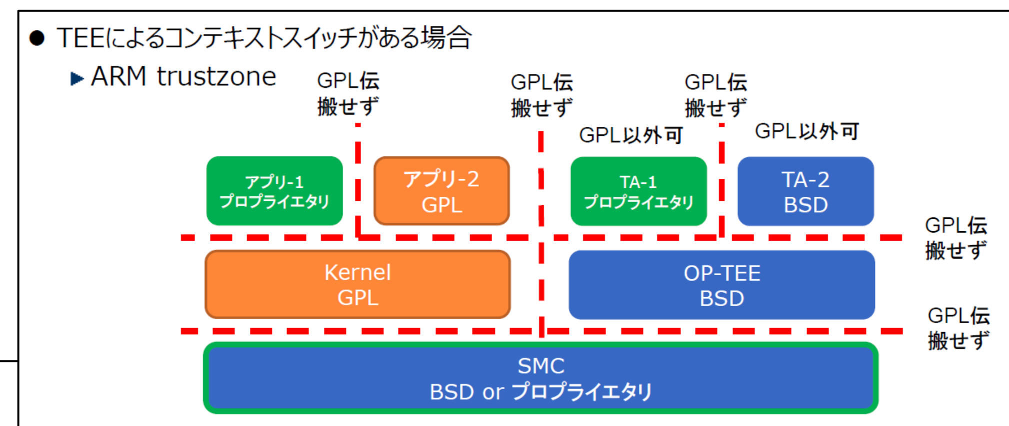
- rpm と deb パッケージを使っている場合
  - Syft, distro2sbom が一般的、本例では distro2sbom
  - インストールされているパッケージがある distribution (ubuntu) の上で実行

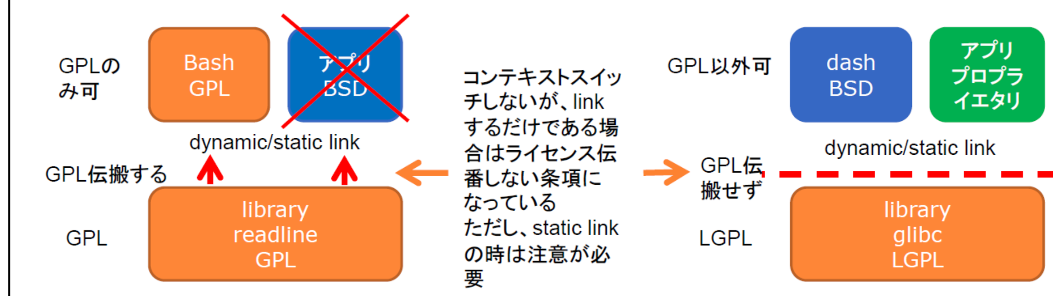
```
$ distro2sbom --distro deb --system --format json --sbom spdx --output-file  ubuntu22-sbom-onsystem.json
```

SBOM の dependency 情報の背景

- Dependency 情報により、オープンソースライセンスに違反せずにソースコード非公開部分を実現
  - 開発時にソースを公開したくない部分の設計に必要
    - 多くの企業は競合他社に対して製品を差異化するためにソースコードを公開しない部分と、ソースが公開されたコンポーネントを組み合わせて、製品化やサービス構築を行っている
    - この時に、GPL 等のソースコードを公開を強制するオープンソースライセンスを使っているコンポーネントとの依存関係を解決していないと、非公開にしたいソースコードを公開する義務が出てしまう。







- https://raw.githubusercontent.com/wiki/ietf-teep/teep-protocol/files/SoftwareLicense-Guide-for-RD-2020-03-30-1.pdf

SBOM ファイルの実情（1/2）

- SBOM という名前が一般的になる前の時代の話
  - セキュリティー以外のバグ修正をトラッキングするため、開発エンジニアはパッケージごとのバージョンを管理していた
  - sed, awk を使ったスクリプトを自作し、ビルドサーバーでビルドするたびに自動的に csv ファイルを吐き出すようにしていた
    - rpm, debian パッケージの場合は spec, dsc ファイルからパッケージ名とバージョンを抽出
    - autoconf を使ったパッケージの場合は configure.ac ファイルの AC_INIT 行からパッケージ名とバージョンを抽出
    - CMake の場合は CMakeLists.txt 内の project(MyProject VERSION 1.0) から抽出
    - 上記のような一般的なビルドシステムを使っていないパッケージは Makefile から抽出
- 問題は、どうしても生成したファイルに足りない項目の情報が出てきて手動でリストを編集する必要があった
  - csv ファイルは手で編集しやすい。excel で開いて見やすい

- 開発時に本リストを保持することにより、特定パッケージにバグが発見された場合、バグ修正された新しいバージョンへ入れ替えや、バグが入っていない古いバージョンに入れ替えが可能になり、更新ファームウェアの作成が容易になる

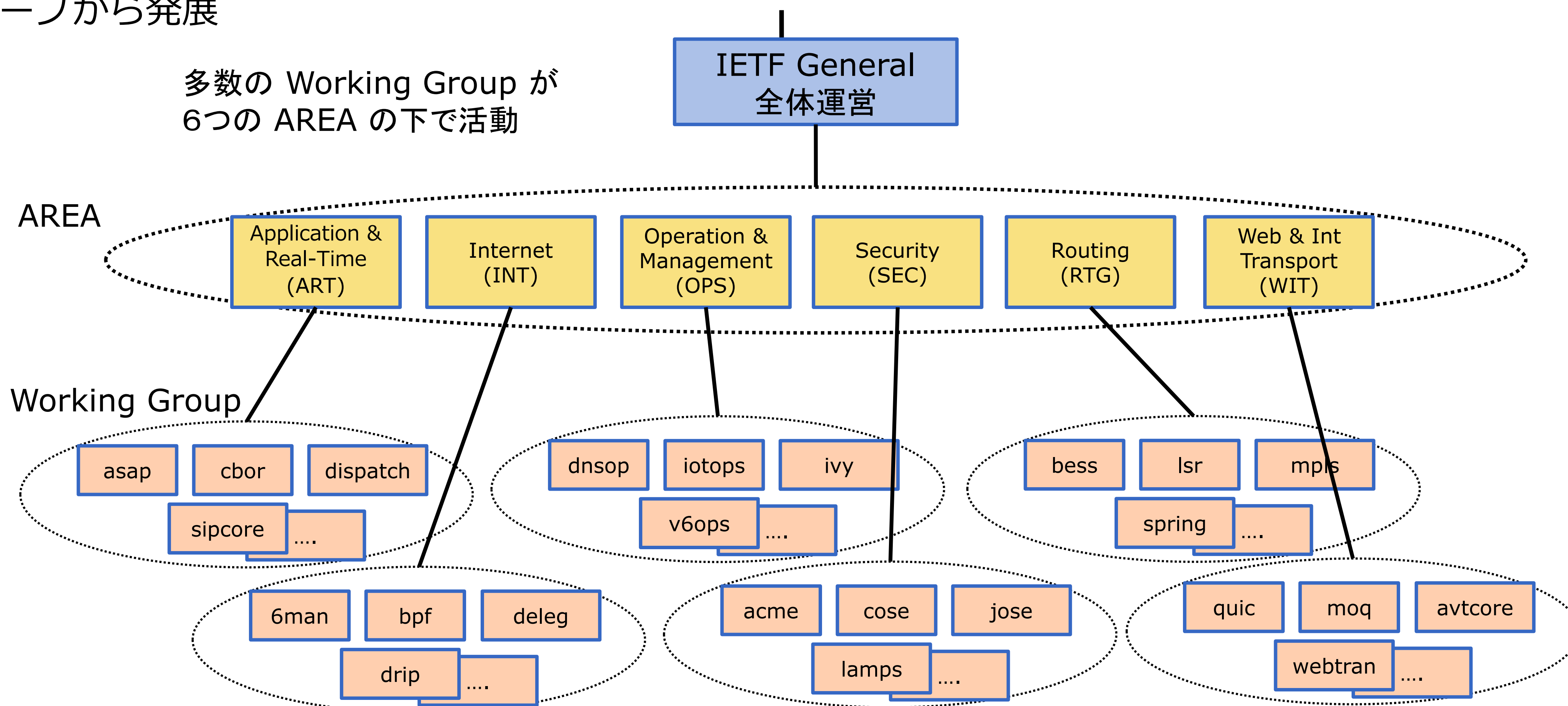
SBOM ファイルの実情（2/2）

- SBOM で多く使われる json は手動で編集しにくい
  - コマンドラインツールの jq, mlr を使い、json を csv に変換し、excel で手動で編集、あるいは sed にて自動修正してから json に戻す
- ビルドサーバーにて SBOM 生成の自動化
  - SBOM を手動で作成する必要があるという認識が一部であるが、上述のように以前よりある程度自動化していた。手動によるミスを避けるためと、業務効率向上の面から、SBOM 生成ツールを組み合わせて可能な限り CI/CD で自動的に SBOM 生成を行う
- セキュリティコンサルは別
  - 出荷する製品に従来より取扱説明書が付属し、各UL, CE, TELEC, PSE 等の安全規格情報を提供していた。
  - それに加えて、今回新たにSBOM リストファイルの提供を追加を義務化しただけであり、購入者が SBOM を解釈してセキュリティ判断を行うのが原則
  - 例として、脆弱性が報告された openssl のバージョンが公開された場合に、購入者が SBOM と照らし合わせて openssl に脆弱性があるバージョンであるかを照合するのが原則である
  - 同時にベンダーの方では、脆弱性が報告された openssl のバージョンを避けたファームウェアを早期に提供が必要
  - 購入者がベンダーに「おたくの製品に使われている openssl に脆弱性を持つバージョンがあるかを教えてほしい」、「公開された openssl の脆弱性を説明してほしい」等の要求がある場合は、SBOM 提供以上の要求になるため、機器ベンダーは対応する必要はない。一般的には専門企業による別途有料セキュリティコンサルのサービスの範疇になる

IETF の概要と組織構造

- IETFとは、インターネット技術の開発と標準化を推進する任意団体

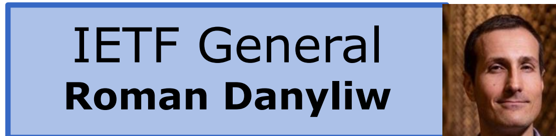
  コンピュータシステム・通信機器・IoT機器を相互接続するための技術開発と標準化文書を議論するグループから発展

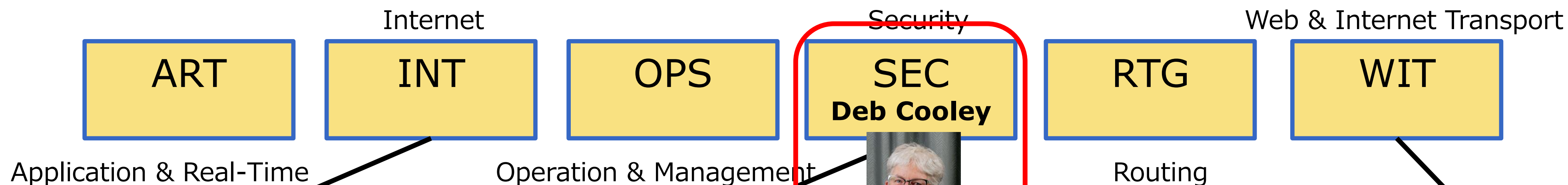


https://datatracker.ietf.org/wg/

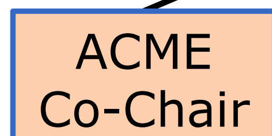
サプライチェーン強化と IETF の組織構造の動き

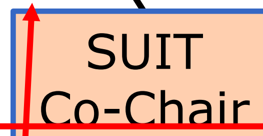
- General の下に７つの AREA がある

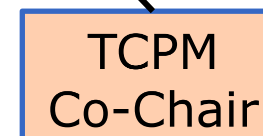




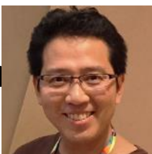
- 各 AREA の下に多数の WG がある



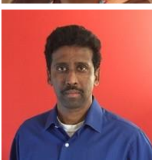




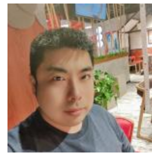
● **Satoru Matsushima**
Softbank



○ **Sri Gundavelli**
Cisco Systems



● **Tomofumi Okubo**
DigiCert　新規



○ **Yoav Nir**
Dell Technologies

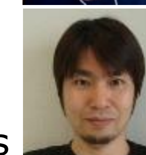
○ **David Waltermire**
NIST -> GSA 職員[注1]

● **Akira Tsukamoto**
ALAXALA Networks
新規

○ **Ian Swett**
Google

○ **Michael Tüxen**
University of Münster

● **Yoshifumi Nishida**
Amazon Web Services



注1：GSA　米国連邦調達庁（General Service Administration）
https://www.soumu.go.jp/main_content/000050144.pdf

米国連邦調達局は、NISTが定める規格を満たす製品のみ調達する
米国はIETF にてサプライチェーンセキュリティーを担保する技術開発と標準化

IETF Chair, Security AREA Director の強化

IETF 119 まで

IETF General Chair
Lars Eggert

→ 御退任

現 Mozilla、元 Nokia CTO
元 Adjunct Professor,
Aalto University フィンランド
元 NEC Laboratories Europe Heidelberg

Security AREA Director
Roman Danyliw

Carnegie Mellon University
Software Engineering Institute,
Deputy CTO

IETF 119 以降

IETF General Chair
Roman Danyliw

Carnegie Mellon University
Software Engineering Institute,
Deputy CTO
セキュリティのボスが IETF のボスに

Security AREA Director
Deb Cooley

NSA を退職されて、SEC Area Director 就任
Commercial National Security Algorithm (CNSA) Suite Profile for TLS and DTLS 1.2 and 1.3 の著者
NIST のコード署名規格にも関係？

セキュリティ強化と流れと IETF 組織改編

- AREA 改編の方針が 118 で発表され、119 で改編完了
  ART と TSV AREA 傘下の Working Group (WG) は、他の AREA に移動

118 までの８つのメジャーな AREA
Applications and Real-Time Area (art) -> 縮小
General Area (gen)
Internet Area (int)
Operations and Management Area (ops)
Routing Area (rtg)
Security Area (sec)
Transport Area (tsv) -> 廃止

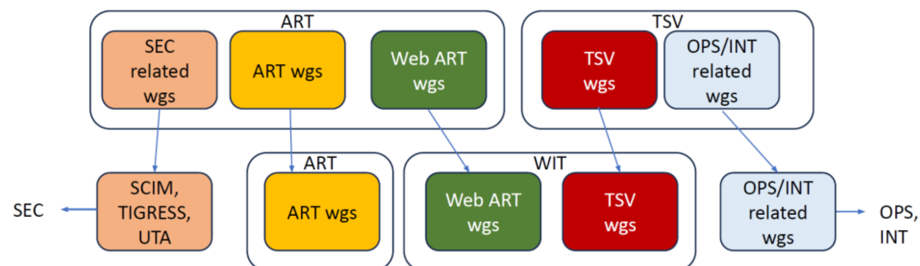
新設
Web and Internet Transport Area (wit)

IETF 119 から

エリア	略称
Application and Real-Time Area	art
General Area	gen
Internet Area	int
Operations and Management Area	ops
Routing Area	rtg
Security Area	sec
Web and Internet Transport Area	wit

表は
https://jprs.jp/related-info/event/2023/IETF118-01.html

ART と TSV に含まれる WG が
他の AREA に移動



Security AREA の他の主な WG の活動内容 (1/2)

日本語にて各 WG の分かりやすい説明がないことで作成。アルファベット順。

- ACE (Authentication and Authorization for Constrained Environments)

  電力、メモリ、CPU処理能力に制限がある組込機器(IoT/Edge)向けの認証と承認の方式を策定。

- ACME (Automated Certificate Management Environment)

  Web サーバー等の X.509証明書の発行、更新、失効を自動化し、HTTPS通信の安全性向上を目指す。

- COSE (CBOR Object Signing and Encryption)

  Concise Binary Object Representation (CBOR)表記にて署名と暗号化を実現するフォーマットの策定。

  CBOR とは組込機器でも扱いやすいよう ASN.1 から簡素化とセキュリティ向上を目指したバイナリーフォーマット。

- JOSE (Javascript Object Signing and Encryption)

  JSON にて署名と暗号化を実現するフォーマットの策定。例、JSON Web Tokens (JWT)

  JSON Web Encryption (JWE), JSON Web Signature (JWS), JSON Web Key (JWK)。

- LAMPS (Limited Additional Mechanisms for PKIX and SMIME

  主にPKIX (Public Key Infrastructure using X.509)、S/MIME について、進化するセキュリティ脅威対する

  堅牢性と、最新の通信需要に効率性の向上に必要な技術の策定。

Security AREA の他の主な WG の活動内容 (2/2)

- RATS (Remote Attestation ProcedureS)
  遠隔サーバーより、対象機器がハッキングされているか等を調査するために必要な、完全性 (Integrity) と真正性（Authenticity）を検証する技術の開発と策定。
- SCITT (Supply Chain Integrity, Transparency, and Trust)
  インターネット通信を用い、サプライチェーンの透明性と信頼性を担保し、攻撃者からの保護とリスクを管理できる技術開発と策定。
- SPICE (Secure Patterns for Internet CrEdentials)
  デジタルクレデンシャル (digital credential) とそのデジタルプレゼンテーション (digital presentation) にて、リンク不可(unlinkability)、選択的開示 (selective disclosure)、修正削除 (redaction) が様々な用途で活用できるデジタルクレデンシャルプロファイル (digital credential profile) を策定。本目的のため RATS, OAuth, JOSE, COSE のフォーマットを拡張する。
- TEEP (Trusted Execution Environment Provisioning)
  遠隔サーバーより、対象機器の Trusted Component (TC, ソフトウェアとデータのこと)のライフサイクル管理(Install, Update, Delete)を行うプロトコルの技術開発と策定。 RATS WG の方式により対象機器がハッキングされているかを検証する。
  サーバーと組込み機器間のバイナリー配布は、SUIT WG のバイナリーフォーマットを使う。
- TLS (Transport Layer Security)
  インターネット通信に必要な暗号通信プロトコルの技術開発と規格策定。

Security AREA 各WG間の連携とユースケース

- 各 WG の概要だけを読むと、個別の課題に取り組んでいるように一見見えるが、実際には WG をまたがって総合的な活動を行っている。

- 各 WG の活動を統合することで、インターネットを用いたデジタル認証と暗号技術により、信用できる認証局(Certificate Authority サーバー)と遠隔の検証サーバーの組み合わせにて、画像等のデータ改竄、OTA 等のソフトウェア改竄、機器改造等の完全性 (Integrity) と真正性（Authenticity）を検証出来るようになる。

- これにより、企業をまたがって次のような使い方が出来るインターネット社会が実現する

  1. 重要な画像・動画（報道用、司法用など）・ファーム等のデータ改竄の発見と防止、元の作成者の識別
  2. 出荷後、世界中に散らばっている機器のソフトウェアのバージョン管理とそのアップデート
  3. 製品に使われている BOM 情報のトレーサビリティの担保
  4. サーバー・通信機器・組込み機器等で使われる証明書・設定データの更新管理
  5. データセンター内で使用するソフトウェアパッケージの改竄防止と、トレーサビリティの担保

SUIT (Software Updates for Internet of Things) とは

攻撃者に狙われる組込 IoT 機器に脆弱性が見つかった時のアップデート技術の標準化

- 活動の目的

  インターネット・USBメモリー等により、安全に組込機器のソフトウェアとデータをアップデートするためのセキュリティ技術の開発と標準化

- 具体的な技術内容

  組込機器のソフトウェアアップデート (OTAだけでなく機能・データの追加含む) に必要な、バイナリーの完全性チェック (Integrity check)、インストール手順の指定等を実現するためのフォーマットを manifest として策定

- 特徴
  - JSON の代わりに、Concise Binary Object Representation (CBOR) を採用
  - CBOR はCPUの違いに依存せず、かつファイルサイズが小さく、低速な CPU と少ないメモリーサイズを採用する組込機器に適する
  - CBOR に署名と暗号化機能を提供する CBOR Object Signing and Encryption (COSE) とインストール手順を指定できることで、対象機器のソフトとデータを安全にアップデートを実現
- SUIT WG のバイナリーフォーマットの有用性
  - RATS (Remote Attestation ProcedureS) と TEEP (Trusted Execution Environment Provisioning) にて SUIT WG のフォーマットを採用

塚本が SUIT WG の Chair になるまでの経緯

- TEEP Protocol の実装行う（2018年の年末開始）
  - 当初 ARM TrustZone 上で実装
  - その後、ARM, RISC-V, Intel SGX の共通 API として Global Platform API のサブセットを 実装し、RISC-V と Intel SGX にTEEP プロトコルの実装を移植（2019年の夏開始）
    https://github.com/ietf-teep/teep-protocol/wiki
  - ドキュメント
    https://github.com/ietf-teep/teep-protocol/wiki/files/teep-device-open/teep-device.pdf
    https://github.com/ietf-teep/teep-protocol/wiki/files/ta-ref-open/ta-ref.pdf
  - ソースコード
    https://github.com/mcd500/teep-device
    https://github.com/mcd500/ta-ref
- TEEP Protocol の Author の一人に（2020年頃）
  https://datatracker.ietf.org/doc/draft-ietf-teep-protocol/
  TEEP にて SUIT のファイルフォーマットを採用
- SUIT WG の Co-Chair に（2024年３月）

国内企業製品のセキュリティーを世界レベルに合わせることと、海外展開を後押しできる立場に

SUIT WG における９月現在の動向

- IETF 119 から 120 間
  - draft-ietf-suit-firmware-encryption-19 (Encrypted Payloads in SUIT Manifests)
    06-07: 塚本 Shepherd Write-Up 書き上げ、IESG for Publication に送る
  - draft-ietf-suit-manifest-26
    07-23: Murray Kucherawy により文章の体裁についてコメント有
    07-25: install の定義番号の変更の修正が承認されたことで、本変更とMurrayのコメントを反映した -27 発行
    Roman へ
- IETF 120 後
  - draft-ietf-suit-firmware-encryption-20 (Encrypted Payloads in SUIT Manifests)
    07/19: IANA Review 終了: ok
    08/12: Ron Bonica: ok
    08/18: Martin Thomson: レビューにコメント有。Encrypted Payloadsの中継者の信頼性が低いときの考慮の記述があってもよいのでは？Russ がそのコメントに対して解説。Hannes 待ち
  - draft-ietf-suit-trust-domains-07（SUIT Manifest Extensions for Multiple Trust Domains）
    07/16: Dave によるレビューにより４点修正依頼
    08/04: Ken より IPR ok 返事有、Brendan からまだ
    08/06: ４点のうち３点修正済み、残りの修正と -08 の submit は Brendan 待ち
    （Deb Cooley さんは、通す気満々、メールのやり取りの雰囲気から）

ハッシュ値計算、署名、暗号化、検証のプログラミング方法 (1/7)

- 本章の目的
  - 本資料にある暗号技術を実務で活用するために必要不可欠プログラミング方法の解説。
  参照リンク（2019年 塚本が書いた (^^)/ ）
  https://github.com/ietf-teep/teep-protocol/wiki/files/ta-ref-open/ta-ref.pdf
- ハッシュ値計算
  - あるデータに対して一意のハッシュ値を計算する。データに更新、データ化け、改ざんがあるとハッシュ値が変わる性質を活用。

```
/* Equivalant of sha3_init() in sha3.c or SHA256_Init() in openssl  */
TEE_AllocateOperation(&handle, TEE_ALG_SHA256, TEE_MODE_DIGEST, SHA_LENGTH);

/* Equivalant of sha3_update() in sha3.c or SHA256_Update() in openssl.
 *
 * It passes only a chunk of data each time.
 * Typically it is used with moving to the next pointer in a for loop to
 * handle large data until the last chunk. Calculating hash value in
 * iteration makes it possible to handle large data, such as 4GB which is
 * not able to have entire data inside TEE memory size and/or only
 * partial data arrives through the Internet in streaming fashion. */
TEE_DigestUpdate(handle, pdata, CHUNK_SIZE);

/* Used combined with the TEE_DigestUpdate.
 * When the data is larger, move to next pointer of chunk in the data
 * for every iteration */
pdata += CHUNK_SIZE;

/* Equivalant of sha3_final() in sha3.c or SHA256_Final() in openssl.
 * This is the last chunk */
TEE_DigestDoFinal(handle, pdata, DATA_SIZE - CHUNK_SIZE, hash, &hashlen);

/* Closing TEE handle */
TEE_FreeOperation(handle);
```

初期化時に、ハッシュアルゴリズムと
ハッシュ値のサイズを指定

本例で行っていないが、一般的には
データを 64byte のようなブロックに分け
最後の final() を呼ぶまで for ループにて
ブロックごとにデータを update() に渡す。
これは、データサイズが update()に渡せる
サイズより大きいことが多いため

最後のブロックを渡すときは update() を
使わずに final() を呼ぶとハッシュ値が
得られる

データが変更されるとハッシュ値が
変わる

ハッシュ値計算、署名、暗号化、検証のプログラミング方法 (2/7)

- 対称鍵暗号アルゴリズムによる暗号化
  - 暗号化する人と複合するする人が、同じ共通鍵を使い暗号化と複合を行う方法。復号も利用方法は同じ

```
/* Generating Key with AES 256 GSM */
TEE_AllocateTransientObject(TEE_TYPE_AES, 256, &key);
TEE_GenerateKey(key, 256, NULL, 0);
TEE_AllocateOperation(&handle, TEE_ALG_AES_GCM, TEE_MODE_ENCRYPT, 256);
TEE_SetOperationKey(handle, key);

// tee_printf("key: ");
// for (int i = 0; i < 256 / 8; i++) {
//   tee_printf ("%02x", key[i]);
// }
// tee_printf("¥n");

/* Prepare IV */
TEE_GenerateRandom(iv, sizeof(iv));

/* Start encrypting test data.
 * Equivalant of EVP_EncryptInit_ex() in openssl  */
TEE_AEInit(handle, iv, sizeof(iv), TAG_LEN_BITS, 0, 0);

/* Equivalant of EVP_EncryptUpdate() in openssl.
 *
 * It passes only a chunk of data each time.
 * Typically it is used with moving to the next pointer in a for loop to
 * handle large data until the last chunk. Encrypting in
 * iteration makes it possible to handle large data, such as 4GB which is
 * not able to have entire data inside TEE memory size and/or only
 * partial data arrives through the Internet in streaming fashion. */
TEE_AEUpdateAAD(handle, pdata, CHUNK_SIZE);

/* Used combined with the TEE_DigestUpdate.
 * When the data is larger, move to next pointer of chunk in the data
 * for every iteration */
pdata += CHUNK_SIZE;

/* Equivalent in openssl is EVP_EncryptFinal() */
TEE_AEEncryptFinal(handle, pdata, DATA_SIZE - CHUNK_SIZE, out, &outlen, tag, &taglen);

/* Closing TEE handle */
TEE_FreeOperation(handle);
```

初期化時に、対称鍵アルゴリズムと鍵長を指定し送信者の共通鍵を作成

多くの暗号化アルゴリズムでは乱数を必要とする
本例では、AES の初期化ベクトル (iv, initialization vector)を生成。
乱数の品質が悪いと脆弱性につながるため真性乱数生成器が要求される

ハッシュ値の計算時と同様、一般的にはデータを 64byte のようなブロックに分け最後の final() を呼ぶまで for ループにてブロックごとにデータを update() に渡す。
これは、データサイズが update()に渡せるサイズより大きいことが多いため

最後のブロックを渡すときは update() を使わずに final() を呼ぶと暗号化されたデータとそのデータのタグが得られる

受信側で共通鍵にて複合することで元データを得られる。タグを比較すること改ざん検出も

ハッシュ値計算、署名、暗号化、検証のプログラミング方法 (3/7)

- 非対称暗号アルゴリズムによる暗号化の利用方法
  - ハッシュ関数と非対称暗号アルゴリズムを使って署名作成と同様な手順の例

```
/* Calculate hash of the test data first */
TEE_AllocateOperation(&handle, TEE_ALG_SHA256, TEE_MODE_DIGEST, SHA_LENGTH);
TEE_DigestUpdate(handle, pdata, CHUNK_SIZE);
pdata += CHUNK_SIZE;
TEE_DigestDoFinal(handle, pdata, DATA_SIZE - CHUNK_SIZE, hash, &hashlen);
TEE_FreeOperation(handle);

/* Dump hash data */
tee_printf("hash: size %d", hashlen);
for (int i = 0; i < hashlen; i++) {
  tee_printf ("%02x", hash[i]);
}
tee_printf("¥n");

/* Generating Keypair with ECDSA_P256 */
TEE_AllocateTransientObject(TEE_TYPE_ECDSA_KEYPAIR, 256, &keypair);
TEE_InitValueAttribute(&attr, TEE_ATTR_ECC_CURVE, TEE_ECC_CURVE_NIST_P256,
                       256);
TEE_GenerateKey(keypair, 256, &attr, 1);
TEE_AllocateOperation(&handle, TEE_ALG_ECDSA_P256, TEE_MODE_SIGN, 256);
TEE_SetOperationKey(handle, keypair);


/* Signing test data.
 * Keystone has ed25519_sign()
 * Equivalent in openssl is EVP_DigestSign() */
TEE_AsymmetricSignDigest(handle, NULL, 0, hash, hashlen, sig, &siglen);

/* Closing TEE handle */
TEE_FreeOperation(handle);
```

署名したいデータのハッシュ値を計算

本例では、ここで非対称暗号アルゴリズムにより送信者の鍵ペアを作成しているが、一般的には送信者が事前に作成。ECDSA P256 アルゴリズムを利用。他に RSA、EdDSA が有名

データのハッシュ値を秘密鍵で暗号化した結果を署名として活用

本署名を送信者の公開鍵にて受信側が複合できれば、送信者の秘密鍵が使われたことを確認でき、確かに送信者により署名されたことを検証できる

データの代わりに公開鍵を証明局の秘密鍵にて署名すると、証明書の作成になる
次ページ参照

ハッシュ値計算、署名、暗号化、検証のプログラミング方法 (4/7)

- それぞれ用途が違うため組み合わせて使われる
- ハッシュ関数　主な用途：データが一致するかの検証

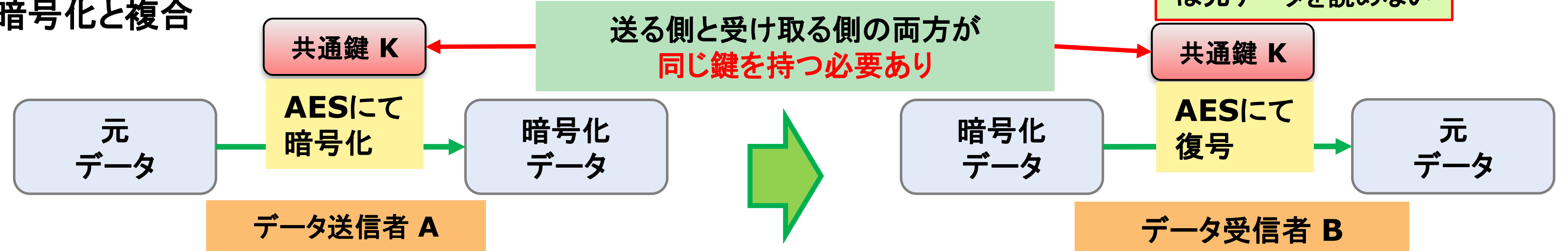
・元データが違えば必ずハッシュ値が変わるアルゴリズムがハッシュ関数
・ハッシュ値を比較することで、元データの改ざんや、ビットエラーを検出できる
・元データーが可変長の時に、元データを比較するのは大変であり、この時に固定長のハッシュ値を比較すればよいことから、ビット化けの検証に非常に便利

- 対称暗号アルゴリズム　主な用途：他人が読めないようデータの暗号化

先に鍵生成

鍵生成すると出来上がる鍵は1つ

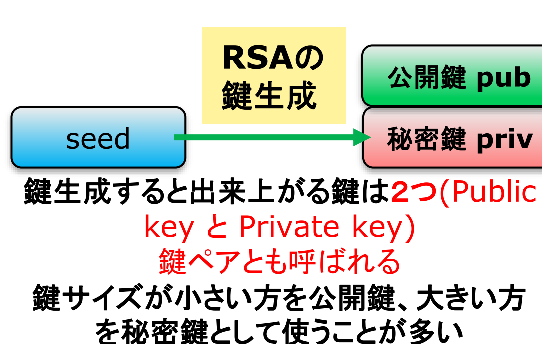
暗号化と複合



共通鍵が漏洩すると、皆が漏洩した鍵で暗号化されたデータを読めてしまう。Bitlocker は本方式

- 非対称暗号アルゴリズム 主な用途：データ作成者特定

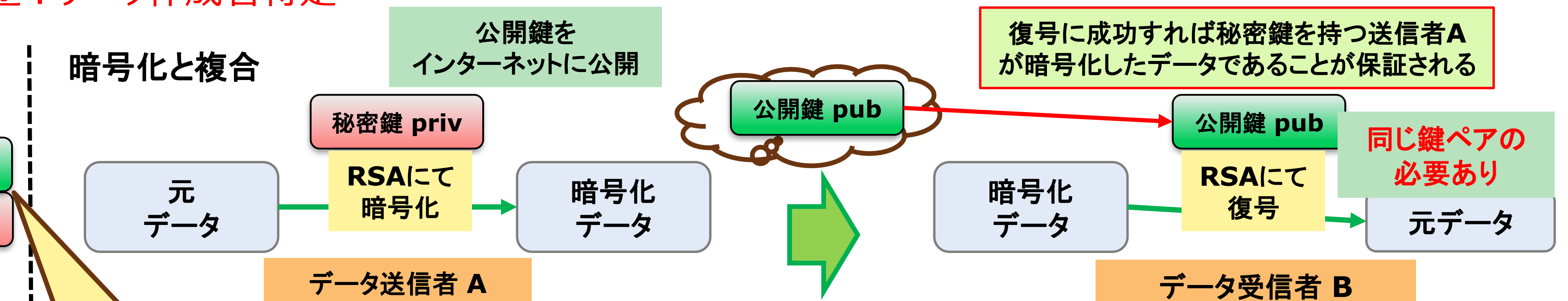
先に鍵生成



鍵生成すると出来上がる鍵は2つ(Public key と Private key)
鍵ペアとも呼ばれる
鍵サイズが小さい方を公開鍵、大きい方を秘密鍵として使うことが多い

同時に作成した公開鍵と秘密鍵のペア以外では復号に失敗する

暗号化と複合



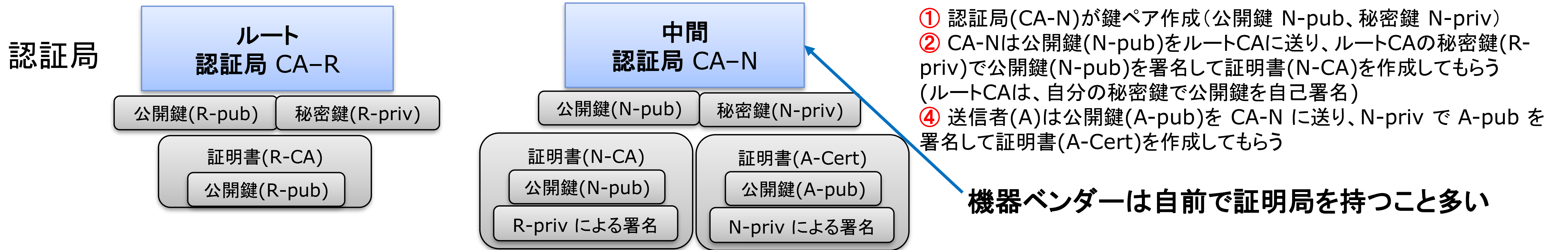
インターネットではサーバーを特定するために使われることが多い

公開鍵を持つものは多数いるためデータの秘匿化には向かないが送信者Aを特定ができる

ハッシュ値計算、署名、暗号化、検証のプログラミング方法 (5/7)

- 証明局、証明書、署名を活用した受信ファイルの署名者検証と改ざん検知の解説

  番号順に前項のプログラミングの使用方法を解説、本例はファームウェアアップデートのバイナリー配布を想定



データ送信

サーバー
送信者 - A

公開鍵(A-pub) | 秘密鍵(A-priv)

事前準備（送信者 A）
③ 送信者Aが事前に鍵ペアを作成
（公開鍵 A-pub、秘密鍵 b-pub）

ファイル送信（送信者 A）
⑧ 送信したいファイルを A-priv にて署名し、この署名をファイル末尾に追加
⑨ ファイルを受信者Bに送信

ファイル（ファームウェア、データ etc）
A-priv による署名

データ受信

事前準備（受信者 B）
⑤ 受信者Bが事前にR-CA,N-CA,A-Certを保持
⑥ N-CAを R-pub にて検証し N-pub が認証局(CA-N)由来であることを担保し、N-pubを抽出
⑦ A-Certを N-pub にて検証し A-pub が送信者(A)由来であることを担保し、A-pub を抽出

機器
受信者 - B

証明書(R-CA)
公開鍵(R-pub)

証明書(N-CA)
公開鍵(N-pub)
R-priv による署名

証明書(A-Cert)
公開鍵(A-pub)
N-priv による署名

ファイル（ファームウェア、データ etc）
A-priv による署名

ファイル受信（受信者 B）
⑩ 受診したファイルを A-pub にて署名検証することで、ファイルが確かに送信者Aが作成され、かつ改ざんされていないことが出来る

ハッシュ値計算、署名、暗号化、検証のプログラミング方法 (6/7)

- 本用途に使われる代表的な暗号アルゴリズム
- ハッシュ関数 hash functions
  - SHA2 - 256/384/512
  - SHA3 - 512
- メッセージ認証コード MAC(Message Authentication Code)
  - HMAC-SHA2 - 256/384/512
  - Poly1305
- 対称暗号アルゴリズム Symmetric Algorithm
  - AES - 128/256
  - ChaCha20
- 認証付き暗号 AEAD(Authenticated Encryption with Associated Data)
  - AES-GCM - 128/256
  - ChaCha20-Poly1305
- 非対称暗号アルゴリズム Asymmetric Algorithm
  - RSA - 3072/4096
  - ECDSA - 256/384/512
  - EdDSA - 256/384/512

- 代表的なルート認証局(root CA)
  - 海外
    - GoDaddy
    - Comodo
    - The SSL Store
    - DigiCert
    - シマンテック（旧ベリサイン）
  - 国内
    - セコムパスポート
    - GMOグローバルサイン
    - JPRS
    - DigiCert
    - サイバートラスト

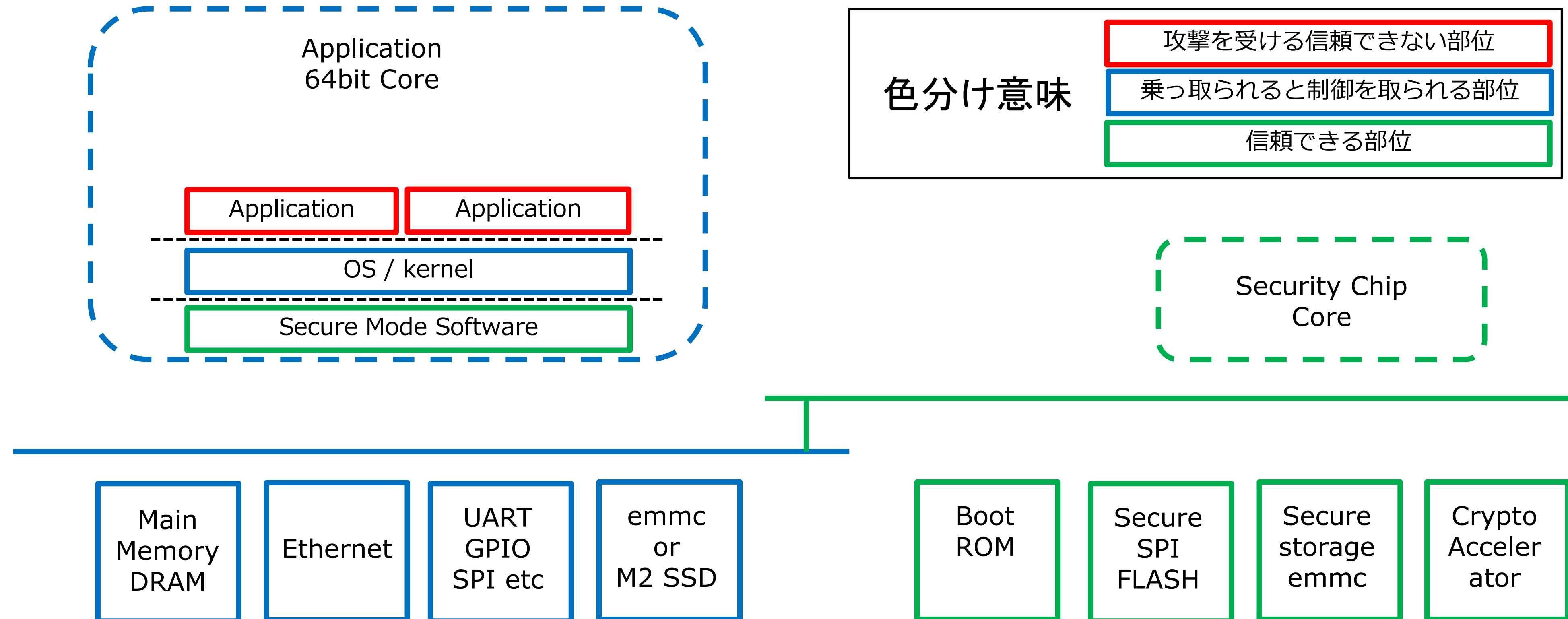
ハッシュ値計算、署名、暗号化、検証のプログラミング方法 (7/7)

- 用途から見たアルゴリズム選択方法

- ファイルがビット化け・改ざんされていかの検証をしたい場合（昔のCRC、チェックサムチェックと同じ用途）
  - ハッシュ関数を使うと元データのハッシュ値と一致するかを比較することで検証可能 <- 一番軽量
  - メッセージ認証コードを使うと元データのタグと一致するかを比較することで検証可能 <- ハッシュ関数より安全性高い
  - 対称暗号アルゴリズムによる暗号化により、複合に失敗することで検証可能（全データを複合する必要有）
  - 非対称暗号アルゴリズムによる暗号化により、複合に失敗することで検証可能（全データを複合する必要有）
- ファイルの中身を他人が読めないようにしたい場合
  - 対称暗号アルゴリズムによる暗号化により、共通鍵を持っている人のみがデータを複合可能
  - 非対称暗号アルゴリズムによる暗号化により、公開鍵を持っている人のみがデータを複合可能
- 暗号化と同時に改ざん検証もしたい場合
  - 認証付き暗号によりファイルを暗号化することで共通鍵を持っている人のみが内容を複合可能である。さらに全データを複合せずともタグが一致するかを比較することでビット化け・改ざん検証可能
- 複数あるファイルのうち、特定のファイルの真の作成者が誰かを保証したい場合。受信したファームが確かに自社作成であり、他社や攻撃者による作成したファームでないことを保証したい場合（指紋認証と呼ばれることも）
  - 非対称暗号アルゴリズムによる暗号化されたファイルを公開鍵によりデータを複合できた場合、その公開鍵に対応した秘密鍵を持った人が確かにデータを暗号化したことが保証されることで、ファイルの真の作成者を検証可能

Root of Trust による Secure Boot 実装例（1/11）

- 本実装例におけるハード例の概要図



Root of Trust による Secure Boot 実装例（2/11）

- ハード実装例

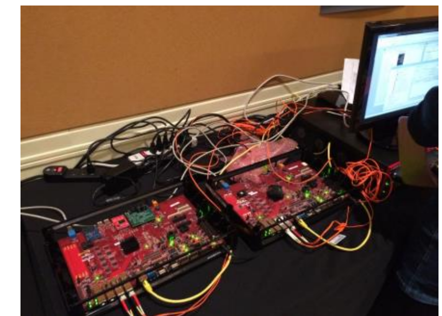
REE / TEE
Rocket/BOOM processor(s)
Data cache
Instruction cache
Isolated sub-system
IBex core
Instruction cache
Isolated Bus (IBus)
RAM
Boot ROM
PUF
ROM / OTP / Flas
System Bus (SBus)
TileLink to AXI4
PCIe controller
L2 cache
Memory Bus (MBus)
TileLink to AXI4
DDR controller
Peripheral Bus (PBus)
GPIO
Boot ROM
SPI (MMC)
UART
SPI (Flash)
Utility group
EC/Ed-DSA
TR
AES-GCM-128/25
RS
SHA3-512
ChaCha20
Pol
HMAC-SHA2
A
Crypto-core gr

https://github.com/uec-hanken/tee-hardware
https://www.mdpi.com/2079-9292/13/13/2508

CN9130 Block Diagram
Cortex-A72 ARMv8 CPU
L1 D-cache 32KB
L1 I-cache 48KB
512KB L2 Cache
1MB L3 System Cache
Aurora2™ Coherency Fabric
SMMU
DDR4 64-bit + ECC DRAM Controller
Integrated BootROM
Secured Boot with OTP
ARM® TrustZone®
Virtualization
Power Management
Thermal Sensor
Low-Dropout Regulator
Real Time Clock
Timers
Security Accelerator
Public Key Accelerator
DMA Acceleration Engines
Management Subsystem
Packet Processor
Parser
Buffer Management
Classifier
PTP (IEEE1588)
Device Bus, NAND, SPI, SDIO3, eMMC5
UART, I2C, GPIO, LED
2 x SATA 3.0
1 x PCIe 3.0 x4
1 x USB 3.0 Device
2 x PCIe 3.0 x1
2 x USB 3.0 Host
10G/5G/2.5G/1G Ports
5G/2.5G/1G Ports
2.5G/1G Ports
32 TDM/VoIP
I2S/SPDIF
SMI/XSMI
2 x USB 2.0 PHY
6 x High Speed SERDES Lanes
2x ICI x4 SERDES

Modem
Cortex-R4
DMA (PL330)
Secure Boot ROM
SRAM
TZMA (BP141)
Generic Interrupt Controller (PL390)
NFC Module
Main Processor
Cortex-A9
L2 Cache (PL310)
AXI Bus
Debug Bus
Trace Port
Debug Access Port
Display Controller
AXI to APB Bridge
S
NS
TZPC
RTC
NV Counter
Timer
OTP Fuse
NS Timer
KMI
DAC
Trace
JTAG + Boundary Scan
Display
SoC Perimeter
ure
Configurable or Security Aware
Keyboard
Mobile

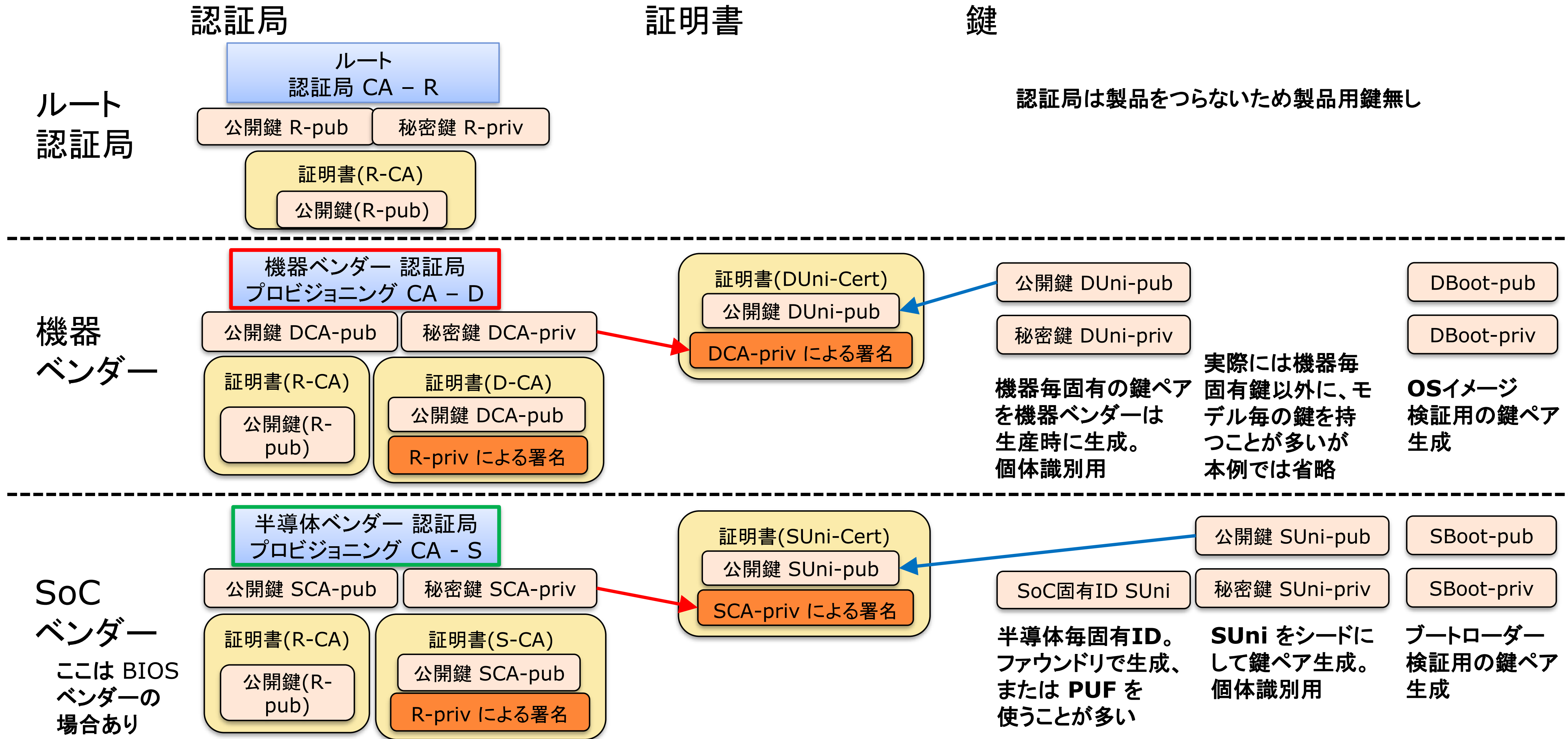
com/developer/article/1169936



https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-infrastructure-processors-octeon-tx2-cn913x-product-brief.pdf

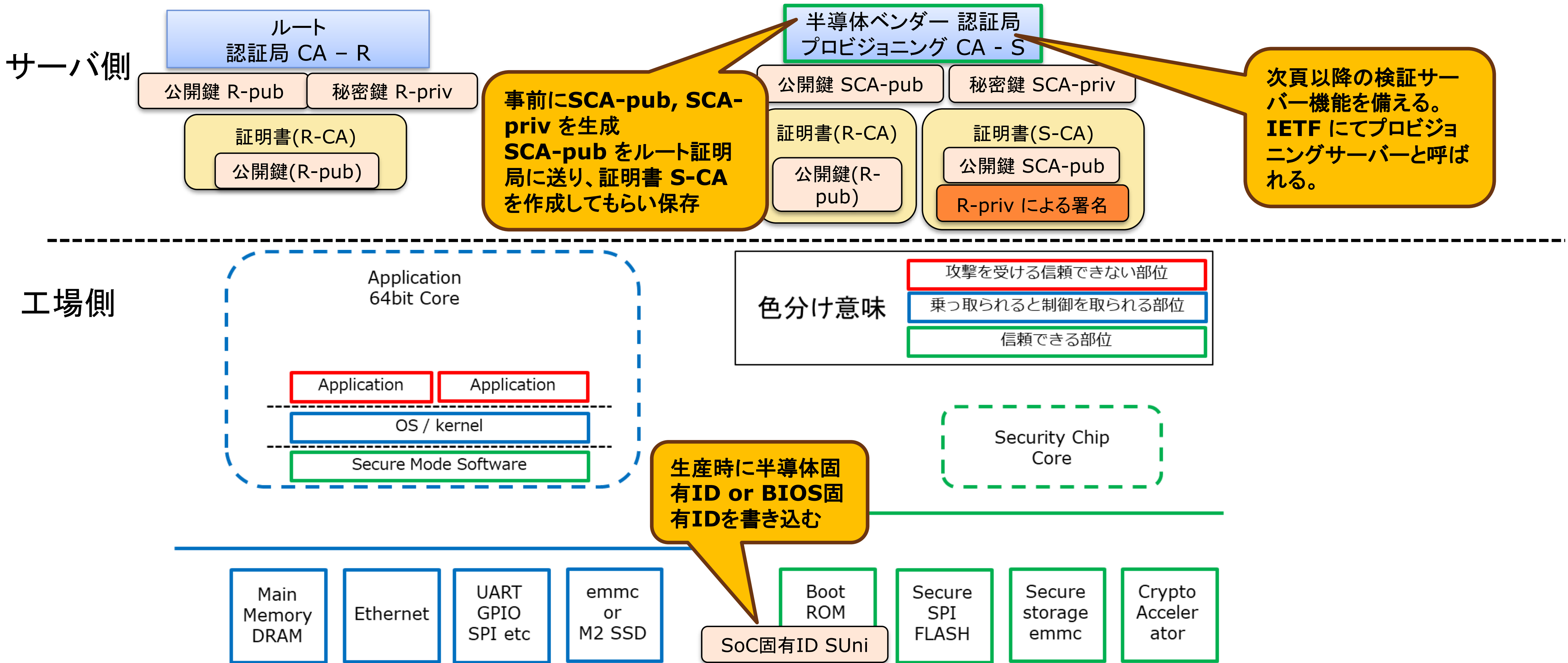
Root of Trust による Secure Boot 実装例（3/11）

- 認証局、証明書、鍵の関係



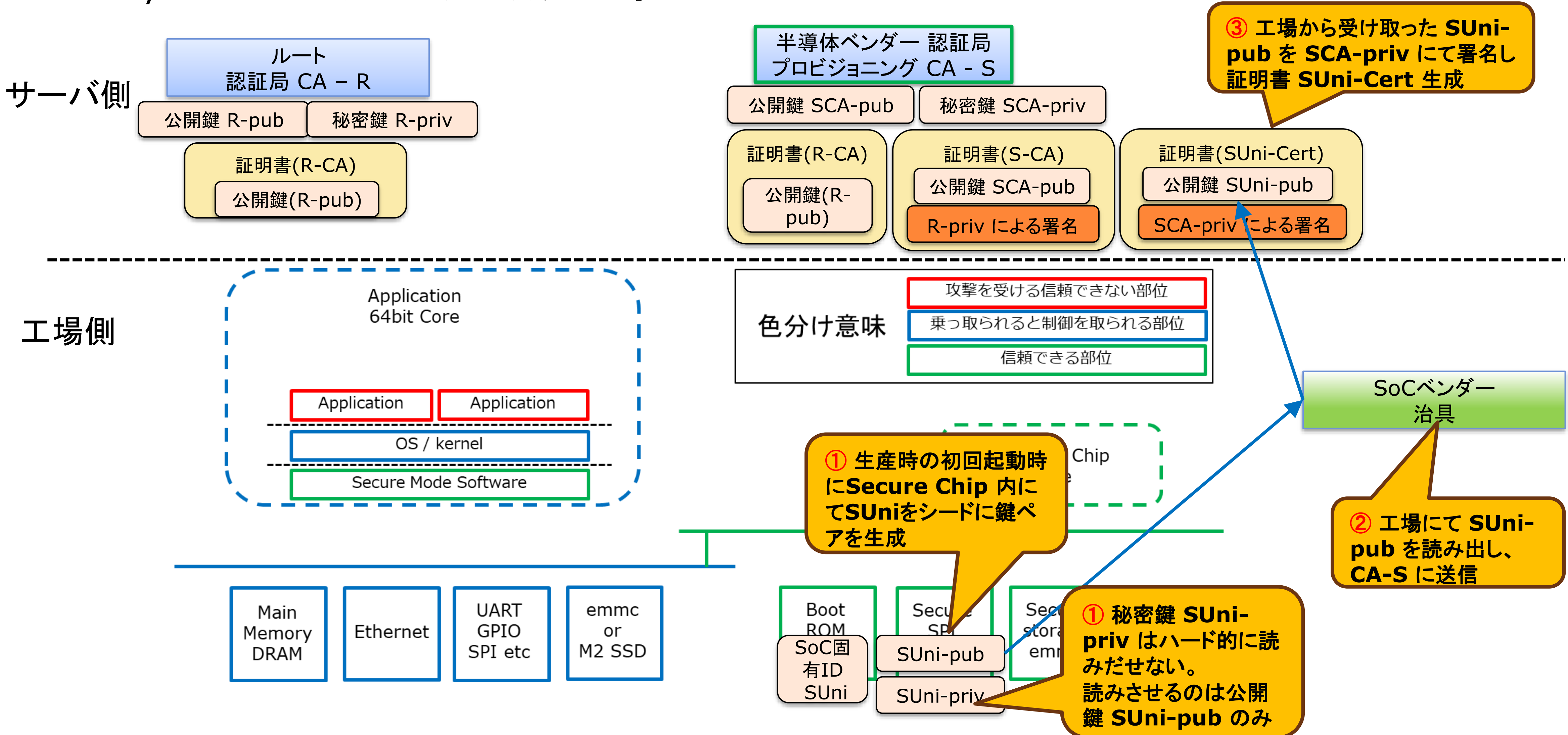
Root of Trust による Secure Boot 実装例（4/11）

- SoC/BIOS ベンダーにおける事前準備



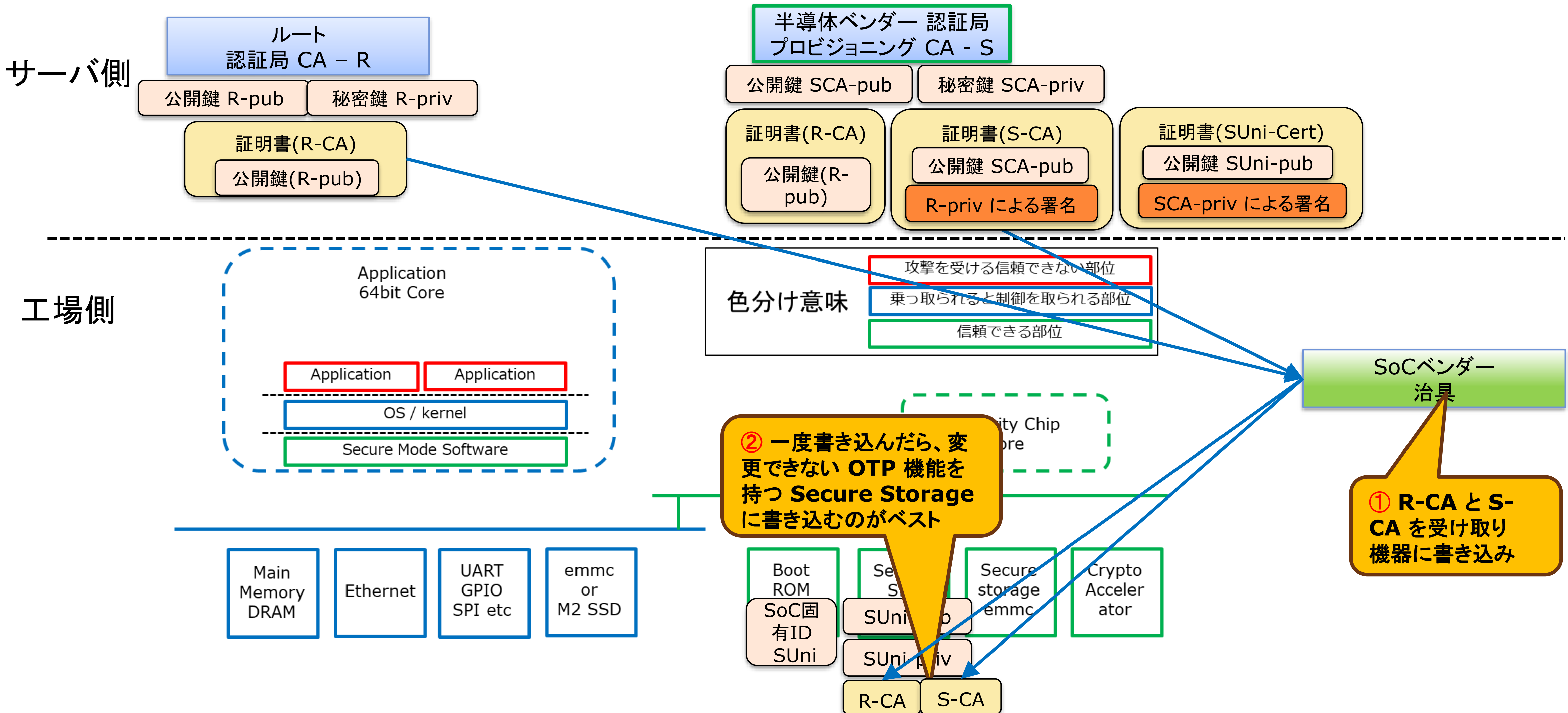
Root of Trust による Secure Boot 実装例（5/11）

- SoC/BIOS ベンダーの鍵生成と登録



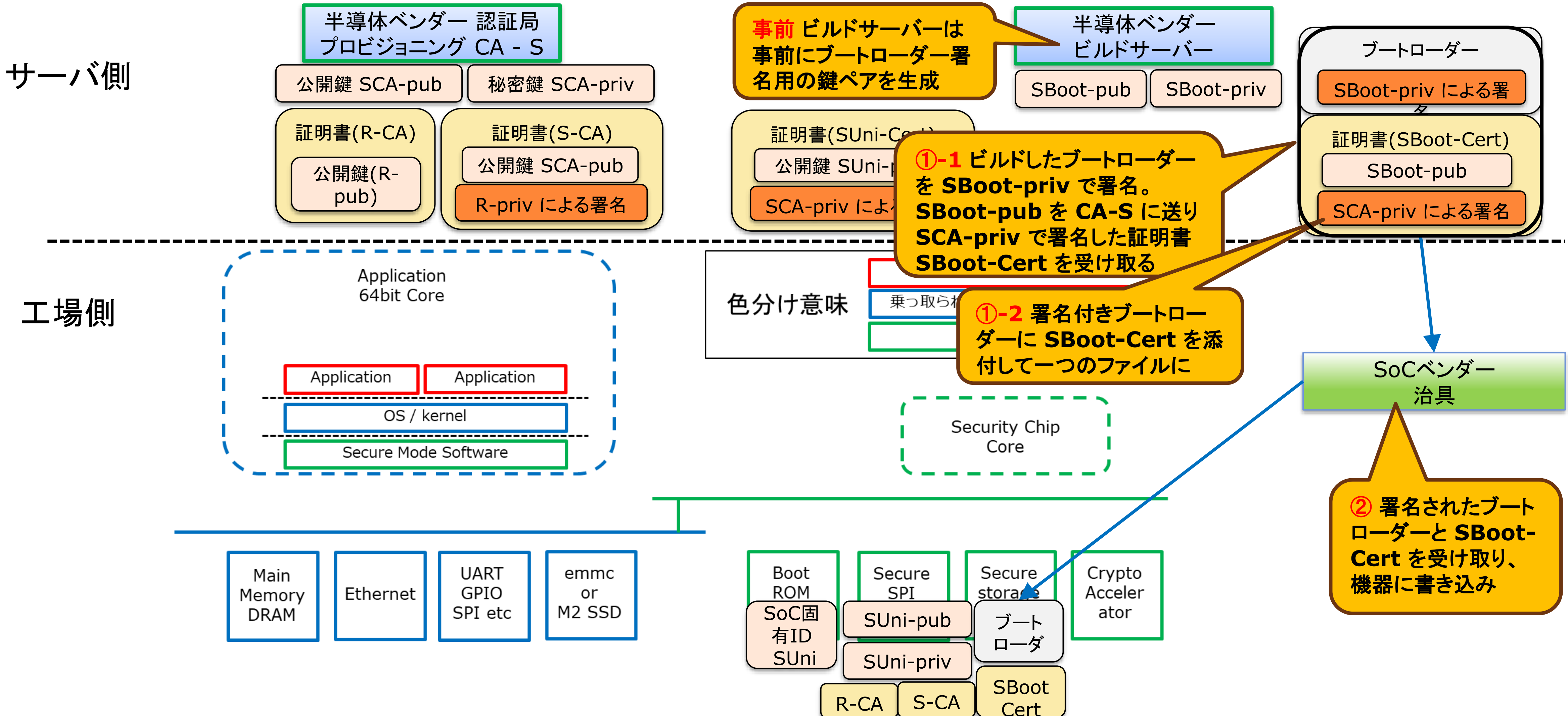
Root of Trust による Secure Boot 実装例（6/11）

- SoC/BIOS ベンダーは、証明書を機器の Secure Storage に書き込み



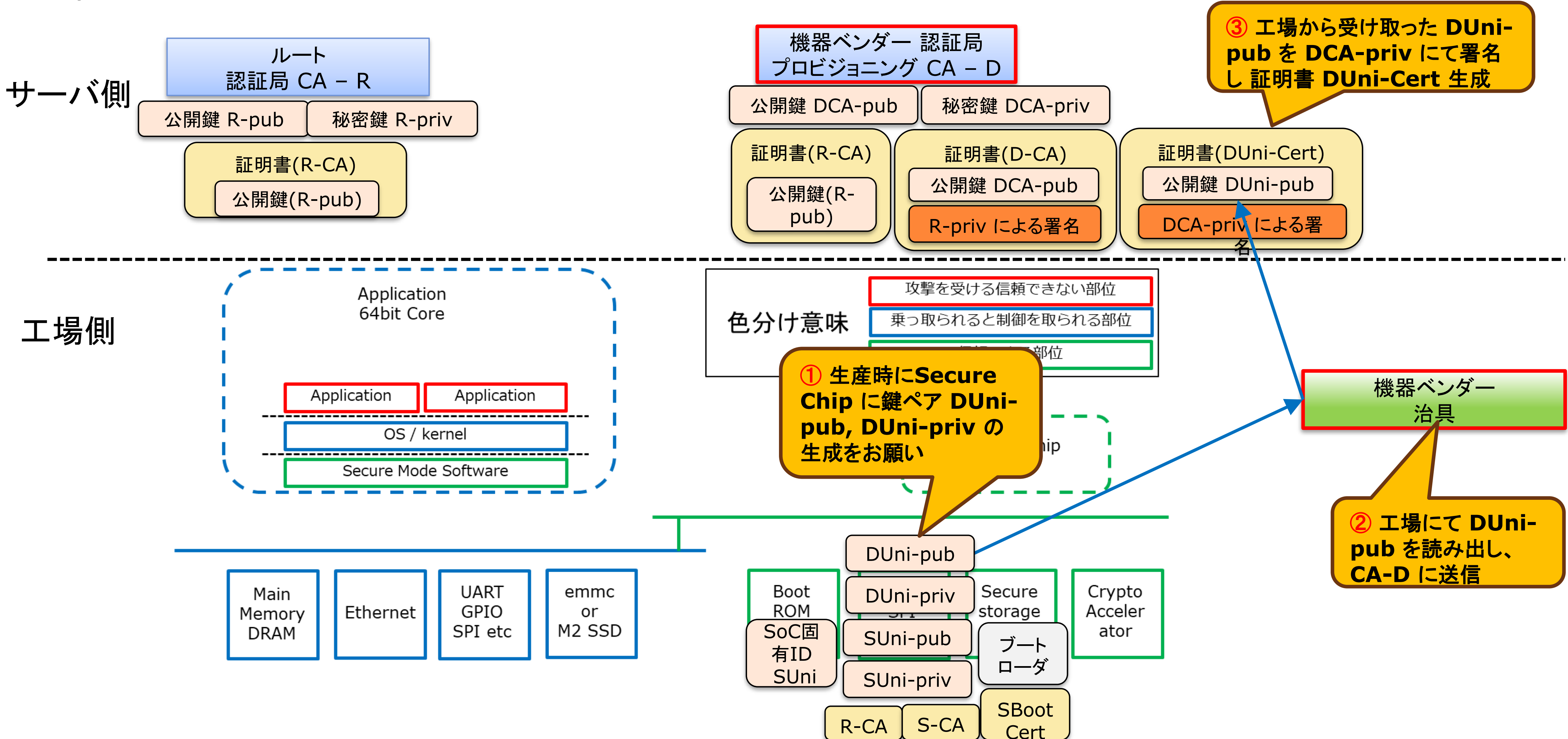
Root of Trust による Secure Boot 実装例（7/11）

- SoC/BIOSベンダー、署名済みブートローダー書き込み



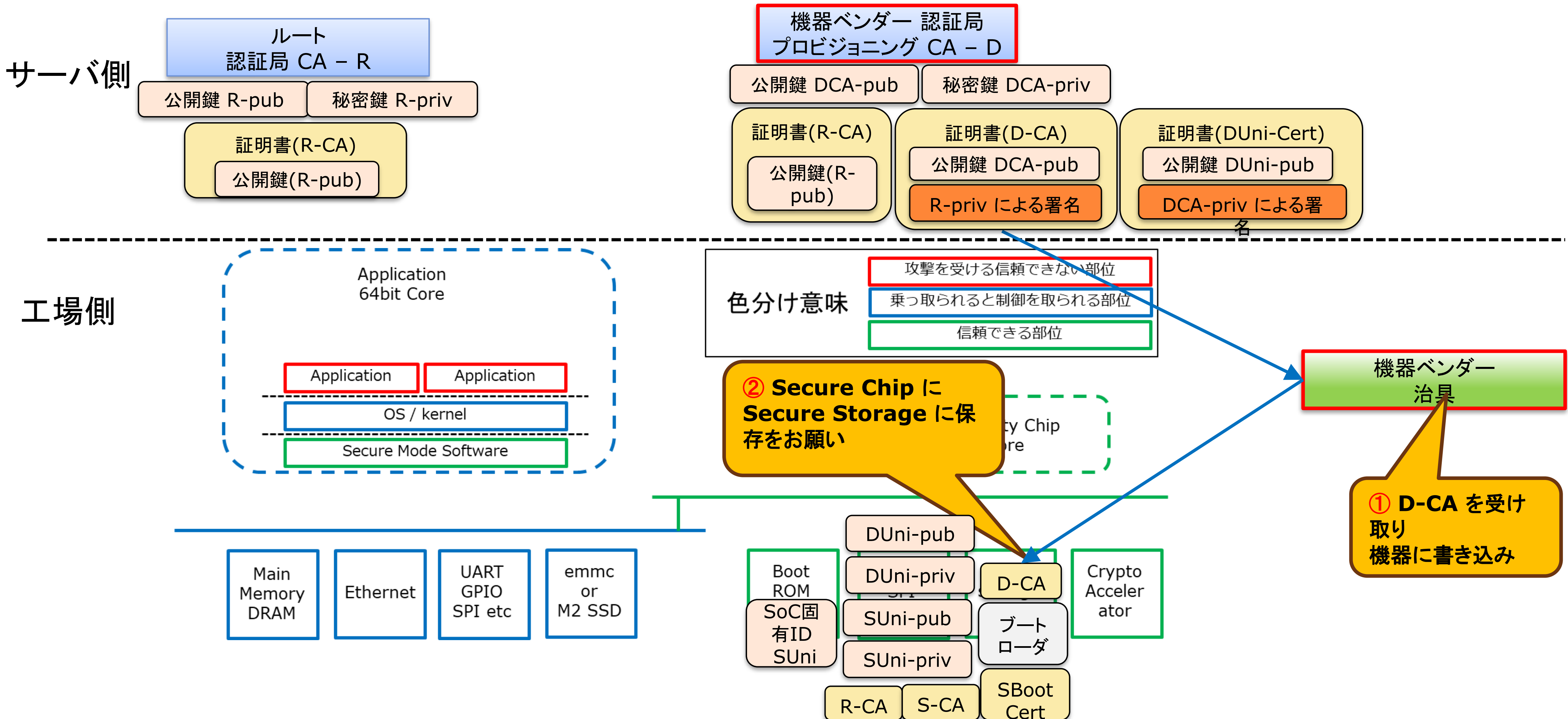
Root of Trust による Secure Boot 実装例（8/11）

- 機器ベンダーの鍵生成と登録



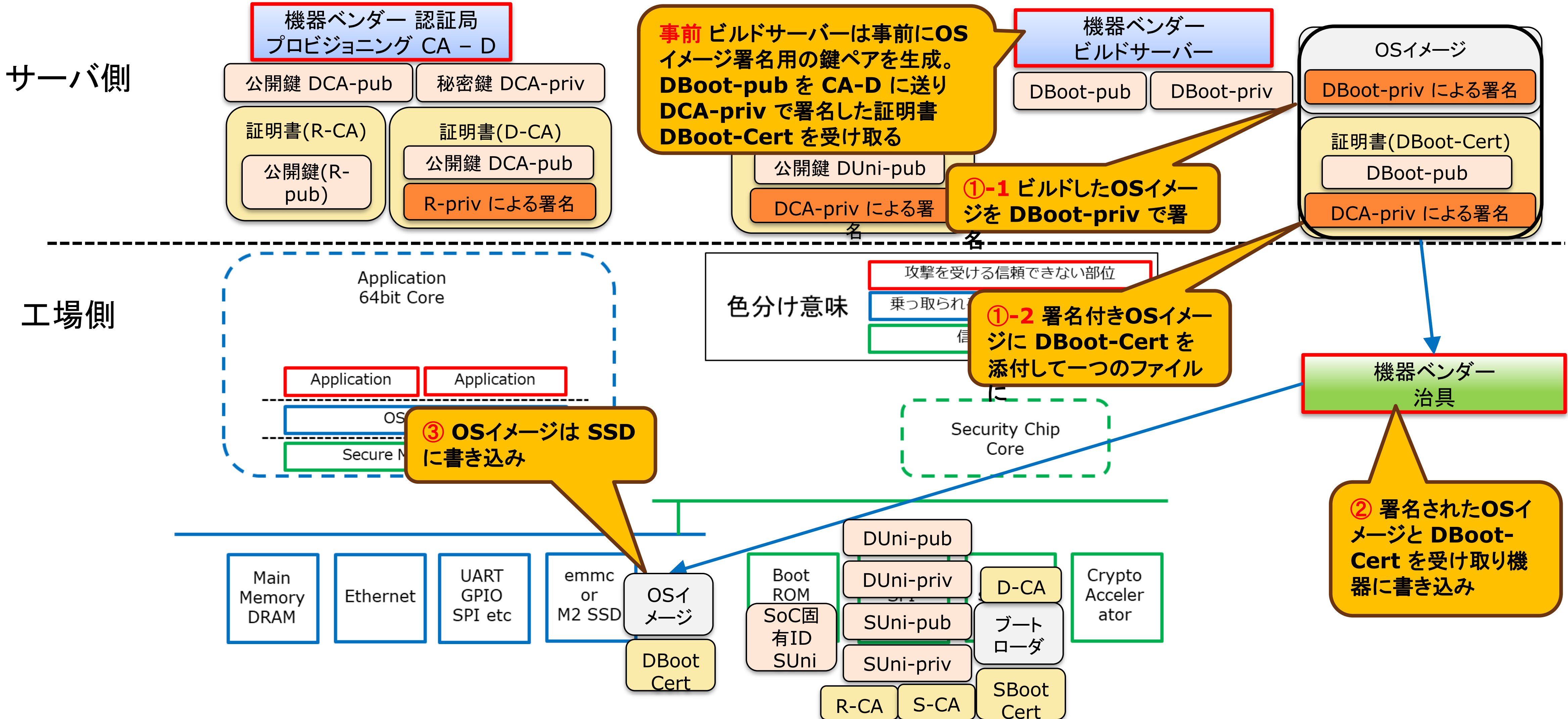
Root of Trust による Secure Boot 実装例（9/11）

- 機器ベンダーは、証明書を機器の Secure Storage に書き込み



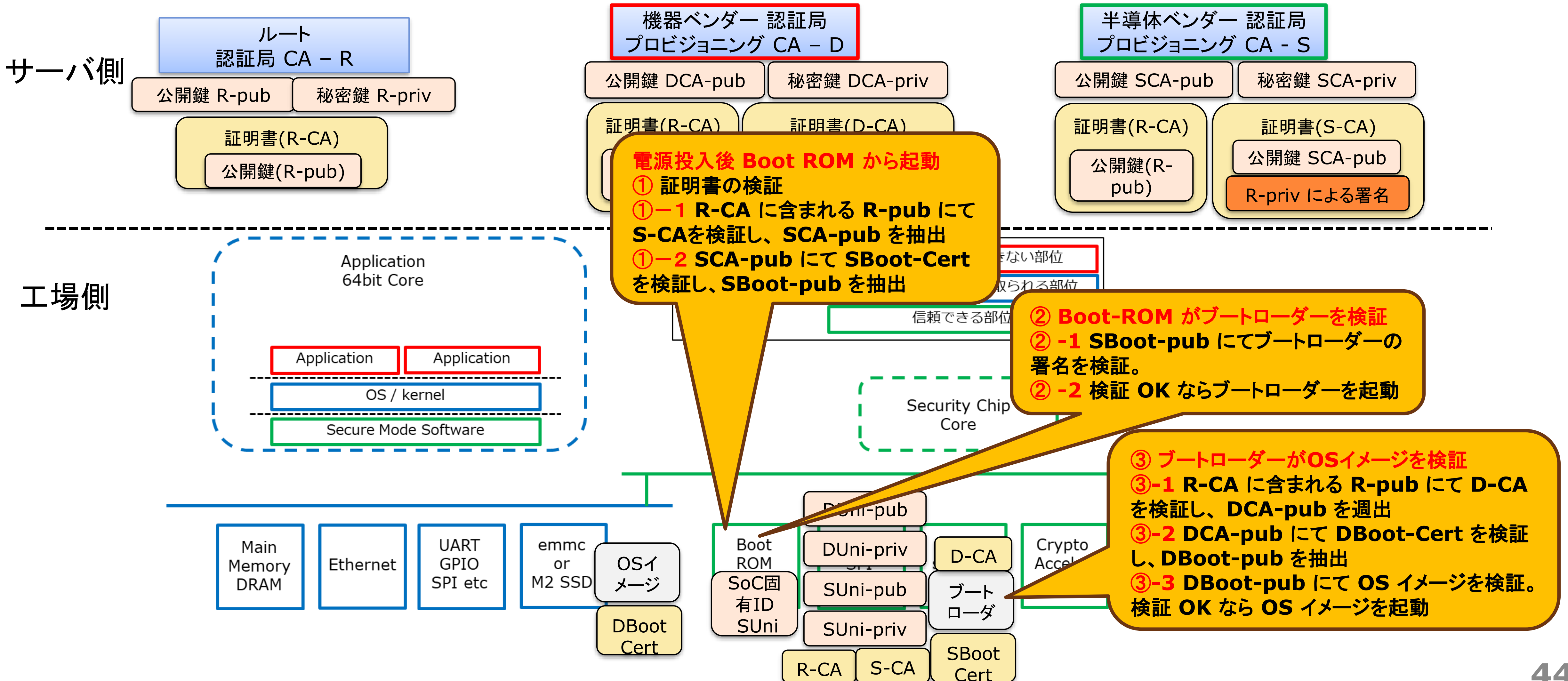
Root of Trust による Secure Boot 実装例（10/11）

- 機器ベンダー、署名済みブートローダー書き込み



Root of Trust による Secure Boot 実装例（11/11）

- 起動時のトラストチェーンの検証手順



組込開発現場で見られるアルゴリズムの選択例

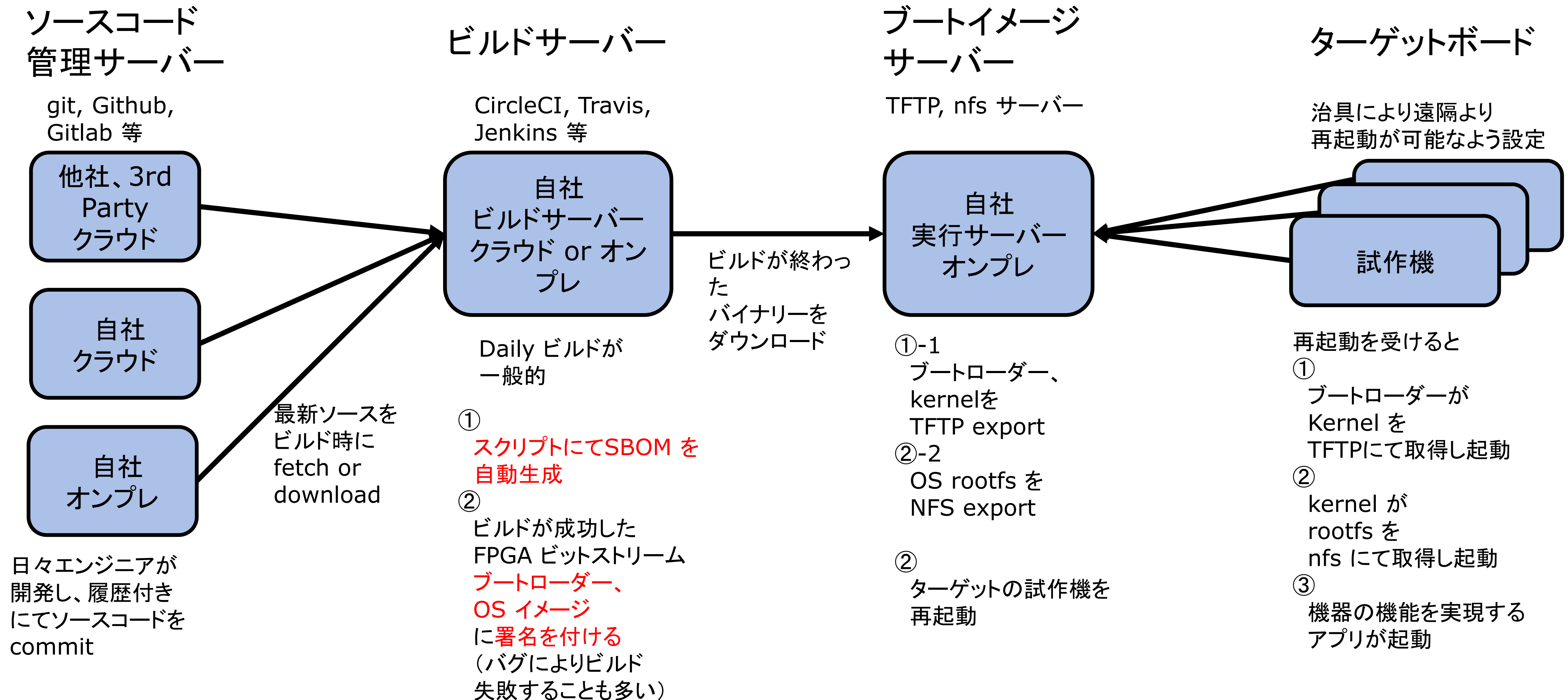
前頁までインターネット業界における暗号化アルゴリズムの使用方法を説明してきたが、目的達成のためによく見られる簡略化例

- ファームウェアのビット化け・改ざん検出、暗号化、作成者認証のすべてが必要な場合
  - 元データのハッシュ値を追加し、対称暗号化アルゴリズムにて暗号化し、そのそのハッシュ値を非対称暗号アルゴリズムにて署名を作成するのが王道だが。。。
  - このファイルを署名検証することで、ビット化け・改ざん検出と作成者認証が同時にでき、そのあとにデータを複合することで目的を達成
  - この方法では暗号ライブラリ（openssl, mbedtls, wolfssl）の複数の API を呼ぶプログラミング（ハッシュ関数、認証付き暗号、非対称暗号アルゴリズムの全てを使わないといけない）が必要になり、プログラマーの工数が高い
  - 非対称暗号アルゴリズム（例 RSA/ECDSA）により全データを暗号化したファイルを作成することにより目的を達することは可能

    ただし遅いです。アルゴリズム的にずっと軽量であるハッシュ関数に頼らずに、全テータを RSAやECDSA にて暗号化して復号するのは 4GHz を超える CPU においても計算パワーが必要で数秒単位の時間がかかる。インターネットの web ブラウザーの通信においては、即座に画面更新が行われることが求められるため、このような使い方はできないが、ダウンロードしたファームウェアを用いたアップデート処理などは、ダウンロード自体に時間がかかっていることから、この処理時間の長さは許容できることが多い。
- ファイルの暗号化と複合の時
  - 数学的には AES 128 より認証付き暗号である AES-GCM 256 を使うことが推奨されているが、 AES-GCM等は内部計算において全データがメモリーに乗っている必要がある実装が一般的であり、メモリー容量が少ない組込機器では使いにくい
  - AES 256 にて元データを 16 KB 等のブロックごとに暗号化と複合の処理を行うことで、必要とされるメモリーの削減が可能
  - ただし、この場合はブロックごとの暗号化と複合のため、このブロックの順番を入れ替えたデータであっても復号に成功してしまい、この復号化したデータを実行するとハングしてしまう
  - そのため、元データのハッシュ値を計算し、このハッシュ値を含めたデータを暗号化し、複合後にハッシュ値が一致するか検証することで、上記を防ぐことは可能

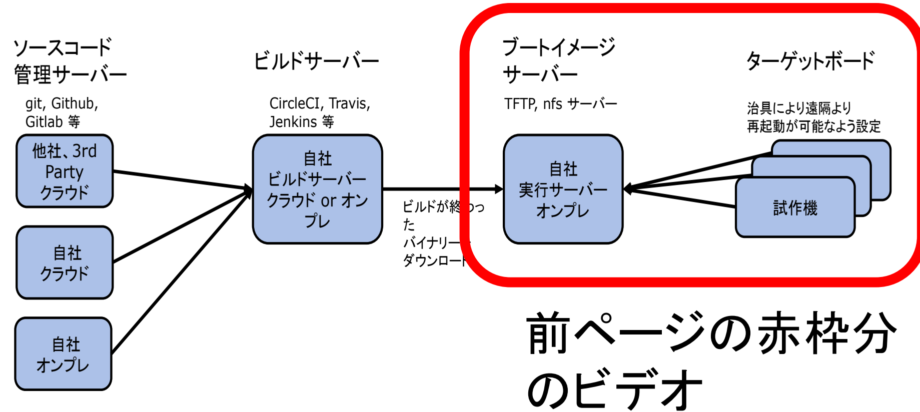
製品開発時の SBOM とファームウェア署名の自動化 (1/2)

- SBOM 生成とファームウェアの署名は、CI/CD にて自動化することが一般的であり、その時の一例

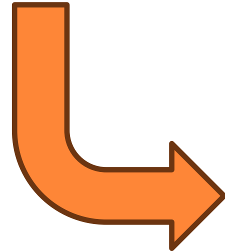


製品開発時の SBOM とファームウェア署名の自動化 (2/2)

- 塚本が前職にて CI/CD を構築した時のターゲット FPGA ボードが起動する動画



前ページの赤枠分
のビデオ



サーバーによるアップデート管理と TEEP の概要

TEEP プロトコルの背景と目的

- 組込機器・IoTデバイスのソフトウェアとデータを最新バージョンに管理すること。この時、IoTデバイスのソフトウェアとパーソナライゼーション・データを更新する前に、ターゲットの機器（IoTデバイス等）が攻撃者によりハッキング（compromised device、侵害されたデバイス）されていないかを、サーバーがリモートで検証するする

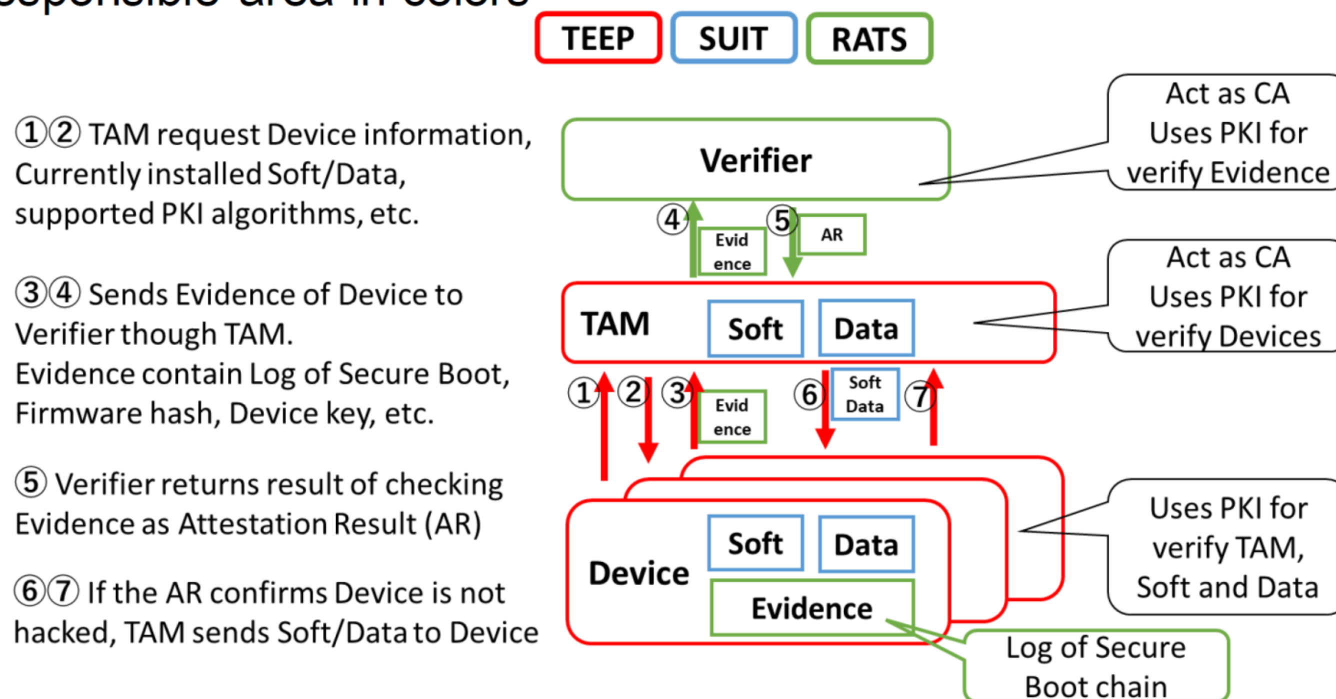
- ターゲット機器とユーザー
ソフトウェアとデータのセキュアアップデートを必要とする、CPUやSoCを搭載した製品を開発するベンダー

- 技術的キーワード
Trusted Applications (信頼できる必要があるソフト) と Personalization data (漏洩してほしくないデータ) のライフサイクル管理を行う
Trusted Applications と Personalization data を合わせて Trusted Component (TC) と呼ぶ

- TEEPは他の WG の成果を活用
TC のファイルフォーマットとして、Software Updates for Internet of Things (SUIT)による manifest を使う。ターゲットの機器が侵害されていないかの仕組みに Remote ATtestation ProcedureS (RATS) を用いる

遠隔サーバーによる機器の健全性の検証とアップデート

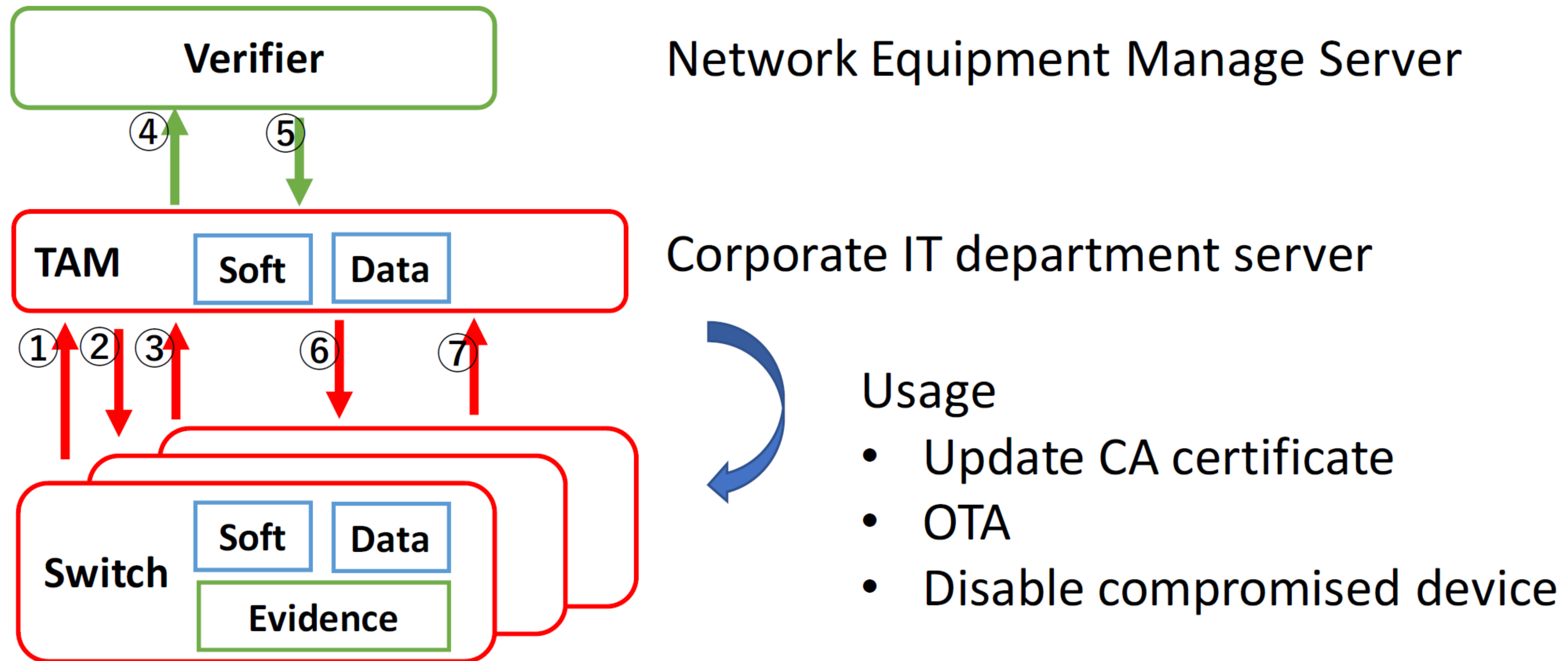
- TEEP による、遠隔サーバーから対象機器がハッキングされないことを検証（Remote ATtestation ProcedureS, RATS）後に、機器のファームのバージョンを把握して適切なアップデート（ライフサイクルマネージメント）
- この時に使われるファイルフォーマットが Software Updates for Internet of Things (SUIT) で定義されている



https://github.com/ietf-teep/teep-protocol/wiki/files/2023-IETF-TEEP-activity-2023-04-18-2.pdf

サーバーによるアップデート管理の活用例

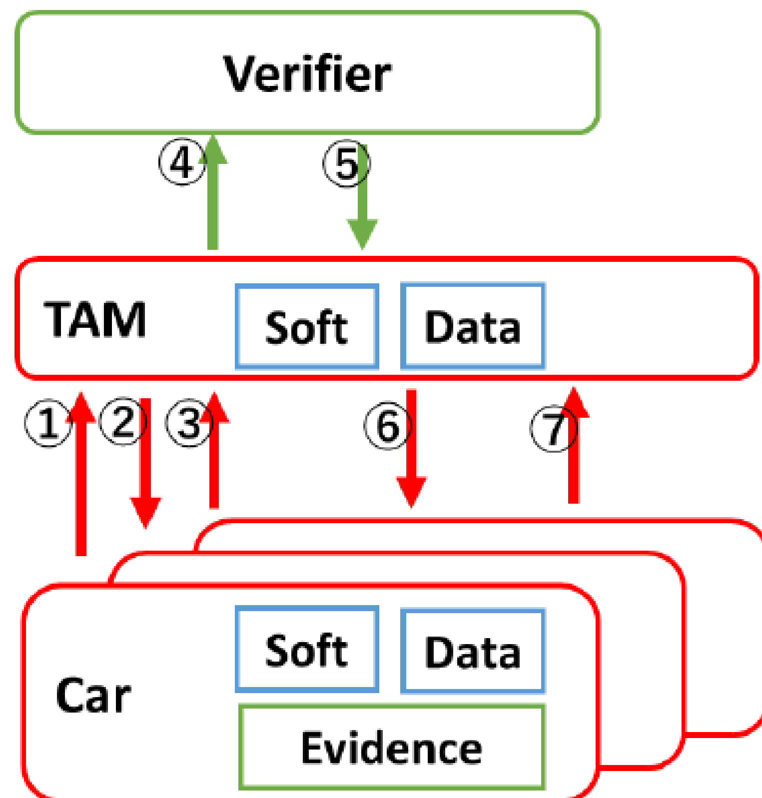
- Network Equipment Vendors



https://github.com/ietf-teep/teep-protocol/wiki/files/2023-IETF-TEEP-activity-2023-04-18-2.pdf

サーバーによるアップデート管理の活用例

- Automotive



Automotive Manufacture

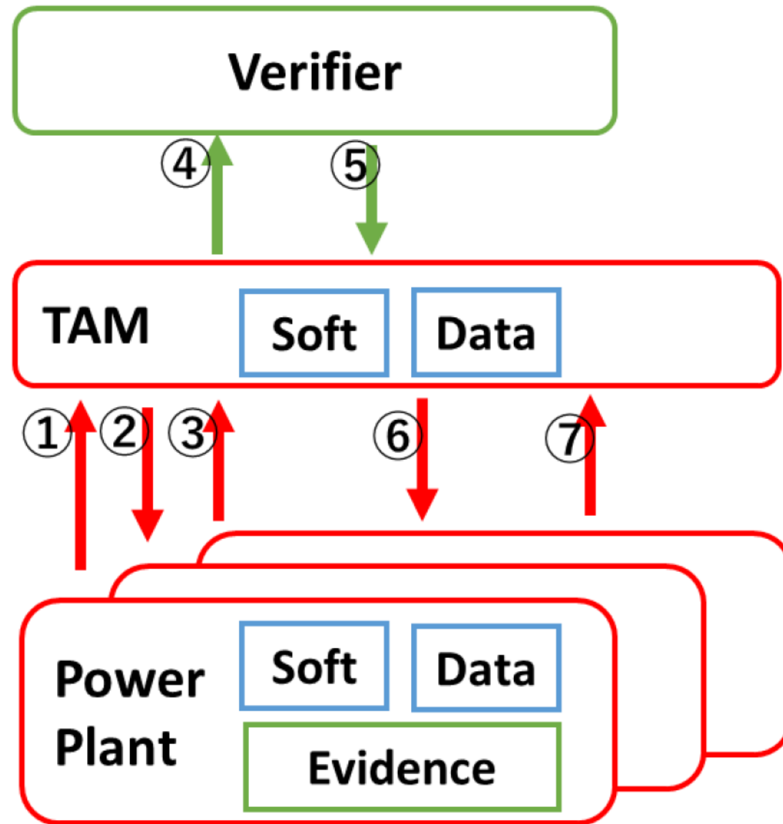
Operators

Usage
- Unlock Quick Charge
- OTA
- Remote monitoring compromised cars
- Remote telemetry acquisition

https://github.com/ietf-teep/teep-protocol/wiki/files/2023-IETF-TEEP-activity-2023-04-18-2.pdf

サーバーによるアップデート管理の活用例

- Electric Power Plants



Power Plant Vendors

Local Government Power Mgmt Server

Usage

- Install security service app for country emergency
- Shut down power plants when a plant is in danger

https://github.com/ietf-teep/teep-protocol/wiki/files/2023-IETF-TEEP-activity-2023-04-18-2.pdf

鍵長と暗号強度の関係（2030年問題）

- 前項のように実世界であ複数の暗号アルゴリズムを組み合わせて使い、組み合わせ時に一つだけセキュリティー強度が弱いのを防ぐ必要がある
- 鍵長によって暗号強度が変わる
  - 日々コンピューターの処理速度が上がることにより解読に必要な時間が短くなることから、使用する暗号強度を年々強くしていく必要あり
  - 2024年現在、一般的には 128 ビットセキュリティがあれば十分。112ビットでは強度が足りない
  - RSA 4096bit 以上は遅く実用性低い
  - IETF 関係者は RSAを使わない人が多く、256ビットセキュリティ以上を使う人多い

表 10　「暗号強度要件（アルゴリズム及び鍵長選択）に関する設定基準」でのセキュリティ強度要件の基本設定方針

ビットセキュリティ	利用上の条件	利用期間		
		2022〜2030 年	2031〜2040 年	2041〜2050 年
112 ビット	新規に処理をする場合	移行完遂期間**	利用不可	利用不可
	過去に処理したものを利用する場合		許容*	
128 ビット	新規に処理をする場合	利用可	利用可	移行完遂期間**
	過去に処理したものを利用する場合			
192 ビット	特になし	利用可	利用可	利用可
256 ビット	特になし	利用可	利用可	利用可

2031年以降使用するのは不可　　2051年以降不可　　2070年まで可

表 11　表 6、表 7、表 8 に記載の暗号アルゴリズム・パラメータに対する「暗号強度要件（アルゴリズム及び鍵長選択）に関する設定基準」での推定セキュリティ強度

セキュリティ強度（ビットセキュリティ）		112	128	192	256
公開鍵暗号（署名・守秘・鍵共有）	素因数分解型（RSA 暗号・RSA 署名）	鍵長 2048 ビッ	鍵長 3072 ビット	鍵長 7680 ビット	鍵長 15360 ビット
	離散対数型（DH(E)、DSA）	鍵長 2048 ビッ (L, N) = (2048, 224	鍵長 3072 ビット (L, N) = (3072, 256)	鍵長 7680 ビット (L, N) = (7680, 384)	鍵長 15360 ビット (L, N) = (15360, 512)
	楕円曲線暗号（ECDH(E)、ECDSA、EdDSA）	P-224 B-233 K-233	P-256 B-283 K-283 W-25519 Curve25519 Edwards25519	P-384 B-409 K-409 W-448 Curve448 Edwards448	P-521 B-571 K-571
共通鍵暗号	ブロック暗号	なし	鍵長 128 ビットの AES、Camellia	鍵長 192 ビットの AES、Camellia	鍵長 256 ビットの AES、Camellia
	認証暗号	なし	なし	なし	ChaCha20-Poly1305
ハッシュ関数	HMAC で使う場合	なし	SHA-1	なし	SHA-256 SHA-384 SHA-512

https://www.cryptrec.go.jp/op_guidelines.html

暗号強度の問題で出荷する製品で使用禁止のアルゴリズム

- 背品寿命が長い製品は、2030年問題の対処が求められる
- 年々コンピュータの計算速度が上がるため、NIST SP 800-52 と IETF にて 2024年 1月 1日以降は使用禁止になったアルゴリズムと代替方法

- ハッシュ関数 hash functions
  - MD5, SHA1

  -> SHA-224以上、SHA3
- 対称暗号アルゴリズム
  - DES, 3DES, RC2, RC4

  -> AES, ChaCha20
- 認証付き暗号 AEAD
  - AES-CBC

  -> AES-CCM, AES-GCM
- 非対称暗号アルゴリズム
  - 3072 bit未満の RSA、256 bit未満の ECDSA、256 bit未満の EdDSA

  -> 256 bit以上の ECDSA、256 bit以上の EdDSA

組込機器における最小限実装すべき暗号方式の規定 (1/2)

- サーバーと違い、CPU・メモリーに制限がある組込み機器で最低限必要とする暗号アルゴリズムの策定
- SUIT-MTI とは Mandatory-to-Implement Algorithms for Authors and Recipients of Software Update for the Internet of Things manifests (SUIT) の略称

- Symmetric: 対称鍵暗号アルゴリズム
  sha256-hmac-a128kw-a128ctr
- Current Constrained Asymmetric MTI Profile 1: 非対称鍵暗号アルゴリズム（低速 CPU 用）
  sha256-es256-ecdh-a128ctr
- Current Constrained Asymmetric MTI Profile 2: 非対称鍵暗号アルゴリズム（低速 CPU 用）
  sha256-eddsa-ecdh-a128ctr
- Current AEAD Asymmetric MTI Profile 1: 認証付き非対称鍵暗号アルゴリズム
  sha256-es256-ecdh-a128gcm
- Current AEAD Asymmetric MTI Profile 2: 認証付き非対称鍵暗号アルゴリズム
  sha256-eddsa-ecdh-chacha-poly
- Future Constrained Asymmetric: 非対称鍵暗号アルゴリズム（将来の耐量子暗号用としての案）
  sha256-hsslms-a256kw-a256ctr

組込機器における最小限実装すべき暗号方式の規定 (2/2)

- AES, ssh 脆弱性にまつわる SUIT-MTI の更新タイミング（当時未公表情報含む）

- 2023/07/31
  aes128-ccm を削除
- 2023/09/02
  aes128-gcm を削除し aes128-ctr に変更
- 2023/10/23
  aes128-gcm と chacha20_poly1305 を追加（これで gcm は復活）
- 2023/11/23 IETF 118 Prague
  aes-ccm と aes-gcm の脆弱性が発表
  （無理な要約をすると ctr と cbc と同等なセキュリティレベルになる）
  - https://datatracker.ietf.org/meeting/118/materials/slides-118-lamps-attack-against-aead-in-cms-00
- 2023/12/20 ssh プロトコルに Terrapin Attack という脆弱性が公開 (tls プロトコルでは脆弱性なし)
  ssh で cbc を使っているときの脆弱性。代わりに aes-gcm を使う方向に

付録

ご参考　耐量子暗号アルゴリズム対応に向けて

- 耐量子暗号アルゴリズムに対応するための活動は、各 WG にて2023年ぐらいから始まっている。
  前頁の暗号強度を強くしても量子コンピューターにおいては実用的な時間にて解読可能であり、
  量子コンピューターにても解読困難な耐量子暗号アルゴリズムが必要とされている
- その二つの対応方法について、技術的ではないが分かりやすい説明
  1. 耐量子暗号アルゴリズムは署名のサイズが非常に大きいことから、通信機器側が、大サイズかつ可変長の署名がついたパケットを受け取っても、従来機器が誤動作しない方法の模索
  2. 従来の暗号アルゴリズムと耐量子暗号アルゴリズムの二つを持った各証明書と署名方式(二つあり Parallel と Composite)のフォーマットと運用用方法の標準化

Summary of existing mechanisms

	LAMPS CMS	OpenPGP	JOSE / COSE
Unlinked Parallel Signatures	✓ RFC5652	✓ RFC 4880	✓ RFC 7518 RFC 9052
Counter-signatures	✓ RFC5652	✓	✓ RFC 9338
Linked Parallel Signatures	✓ RFC5752		
Composite Signatures	I-D exists	I-D exists	

SLH-DSA aka SPHINCS+

Parameter set	# of sigs	Private size	Public size	Signature size
SPHINCS+-128s	2^{64}	64 bytes	32 bytes	7856 bytes
SPHINCS+-128s-q20	2^{20}	64 bytes	32 bytes	3264 bytes
SPHINCS+-192s	2^{64}	96 bytes	48 bytes	16224 bytes
SPHINCS+-192s-q20	2^{20}	96 bytes	48 bytes	7008 bytes
SPHINCS+-256s	2^{64}	128 bytes	64 bytes	29792 bytes
SPHINCS+-256s-q20	2^{20}	128 bytes	64 bytes	12640 bytes
RSA-2048	∞	~256 bytes	~256 bytes	256 bytes
ECDSA-P256	2^{128} (~∞)	~32 bytes	~64 bytes	64 bytes

https://datatracker.ietf.org/meeting/119/materials/slides-119-pquip-composite-vs-parallel-signatures-comparison
https://trustedcomputinggroup.org/wp-content/uploads/TCG-JRF_Workshop-2024-Event-Program_EN_02-26-2024.pdf