Proof of Spaceを活用した改ざん検知システムの提案

伊藤 真奈美[1,a)]　山越 公洋[1]　瀧口 浩義[1]　中津留 毅[1]

概要：近年，IoT の普及等により，組み込みシステムにおいてセキュリティ対策の重要性が高まっている．一方で，組み込みシステムはコスト等の観点から，耐タンパ性を持ったハードウェアを組み込むことが難しく，サーバ等に比べて Root of Trust の確保が困難である．そのような機器で改ざん検知をセキュアに実施するためには，ハードウェア的な Root of Trust に頼らない改ざん検知技術が必要となる．本稿では，ハッシュ比較による改ざん検知をベースとし，Proof of Space を活用して，不正ソフトウェアの追加等の攻撃に耐性を持つ改ざん検知システムを提案する．

キーワード：Remote Attestation, Proof of Space, 改ざん検知

Tamper detection system using Proof of Space

Manami Ito[1,a)]　Kimihiro Yamakoshi[1]　Hiroyoshi Takiguchi[1]　Takeshi Nakatsuru[1]

Abstract: Recently, various embedded systems have been connected to the internet as the Internet of Things (IoT) devices. On the other hand, many of these devices do not have any security component, which makes IoT security difficult. In this paper, we propose the tamper detection system without using any security component. Our method is based on the Proof of Space, which is used to prove that the empty space of the device is truly empty. Our method is secure against data substitution attacks and so on.

Keywords: Remote Attestation, Proof of Space, tamper detection

1. はじめに

1.1 背景

近年，Internet of Things (IoT) の概念の普及により，さまざまな組み込み機器がネットワークにつながり始めている．家電や工場，オフィスなど，さまざまな分野で IoT ソリューションが提案され，私たちの生活の変革が期待される一方で，そのセキュリティリスクが懸念されている [1][2][3]．2016 年に web カメラをターゲットとしたマルウェア Mirai によって大規模な DDoS 攻撃が仕掛けられた事件は記憶に新しい [2]．

機器がマルウェア等に犯されておらず，正しい状態であることをシステム運用者が確認する手段として，汎用 PC では一般的にアンチウィルスソフトが利用される．しかし，アンチウィルスソフトはマルウェアの検体をブラックリストとして登録する必要があり，機器が多種多様で，検体となるデータが少ない IoT 機器ではその導入は難しい．そこで，IoT 機器はプログラムファイルがインストール後あまり変更されないことを利用し，機器にインストールされたソフトウェアがあらかじめ定義された正しい状態から乖離しているか否かを，機器の動作に影響を与えない起動時等に検証するホワイトリスト型の検知技術の適用が期待されている．

ホワイトリスト型の検知技術の多くは，機器にインストールされたソフトウェアについて，正しい状態を定義しておき，その定義情報と一致するか否かを検証する方法をとる [4][5][6]．このとき，正しい状態の定義情報や，検証プログラムの出力が改ざんされてしまうと，検知結果は信頼できなくなってしまう．たとえば，チェックサムをマルウェアが混入している状態のチェックサムにすり替えられ

[1] NTT セキュアプラットフォーム研究所
NTT Secure Platform Laboratories, NTT Corporation.
[a)] manami.itou.xh@hco.ntt.co.jp

てしまうと，マルウェアが混入している状態を正しい状態と判定してしまう．そのため，正しい状態の定義情報や，検証プログラムの出力が正しいことを保証する必要ある．

起動時のホワイトリスト型の検知技術として，Secure boot が知られている [4][5]．Secure boot は，検知結果を信頼できるものとするために，Chain of Trust の考え方を採用している．Chain of Trust とは，信頼されたプログラム/データによってあるプログラム/データが検証されて信頼できると判断されると，さらに，その信頼されたプログラム/データが，他のプログラム/データについて信頼できるか否かを検証していくというステップを順番に行い，信頼できるプログラム・データの範囲を広げていく考え方である [7]．

Chain of Trust の考え方では，一番最初に信頼されたプログラム/データとして検証を開始するプログラム/データを Root of Trust と呼ぶ．Root of Trust は Chain of Trust の信頼性の根幹となるため，一般的に耐タンパモジュール等を用いて厳重に保護されている [8]．Secure boot では，TCG が標準化している TPM を Root of Trust として用いた方法が提案されている．

しかし，IoT 機器は単一の機能に特化していて，機能拡張を想定しない設計がなされることが多く，ハードウェアの追加は汎用 PC 以上に困難である．そこで，本稿では，特別なハードウェアを利用しないセキュアなホワイトリスト型改ざん検知技術を提案する．

1.2 ターゲット

本稿ではプログラムおよびデータの改ざんや不正プログラムおよびデータの追加があるか否かを外部の機器が検証できるようにすることを目的とする．モデルの概要を図 1 に示す．本稿では，プログラムおよびデータをまとめてソフトウェアと呼び，ソフトウェアを格納する領域をソフトウェア領域と呼ぶ．

ソフトウェア領域の改ざんおよび空き領域への不正ソフトウェアの追加がないことを確認したい対象の機器を検証対象機器と呼ぶ．また，ソフトウェア領域の改ざんおよび空き領域への不正ソフトウェアの追加がないことを検証する機器を検証機器と呼ぶ．検証機器は，セキュアであることを前提とする．検証機器は検証対象機器における検証したい領域のソフトウェア領域と空き領域について，あるべき状態をあらかじめ定義できることを前提とする．検証機器は検証対象機器のソフトウェア領域と空き領域のあるべき状態の定義情報をセキュアに保持しており，検証に使うことができる．

検証機器には検証用サーバプログラムがインストールされており，検証対象機器にインストールされている検証用クライアントプログラムに，ソフトウェア領域に改ざんがなく空き領域に不正なソフトウェアが追加されていない状態でなければ正しい response が不可能な challenge を要求し，検証対象機器の検証を行う．

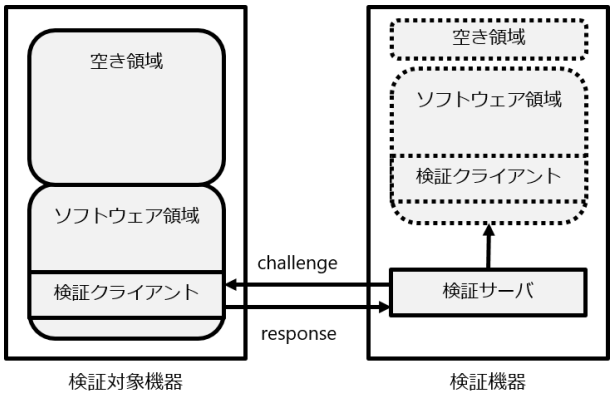

図 1 モデル
Fig. 1 Model.

1.3 攻撃モデル

攻撃者は検証対象機器において，全ての領域にアクセスし，読み書き可能な権限を持つことを前提とする．すなわち，攻撃者は検証対象機器のソフトウェア領域の内容の書き換えや消去，空き領域への不正ソフトウェアの追加を行うことができる．ただし，攻撃者は検証対象機器において，メモリ容量を増やす等のハードウェアの改ざんはできないことを前提とする．

本稿のモデルでは，あるべき状態の定義情報を検証機器がセキュアに保持しているが，検証用クライアントプログラムは改ざんされることを想定する．

1.4 要件

ソフトウェア領域の改ざんや空き領域への不正ソフトウェアの追加がないことを証明するには，ソフトウェア領域が改ざんされていないことと，空き領域が確実に空いていることを同時に証明することができればよい． 本稿における要件を以下のように定める

1) インストールされているソフトウェアが改ざんされていないことを高い確率で証明可能であること．
2) ストレージの空きスペースに不正なソフトウェアが追加されていないことを高い確率で証明可能であること．
3) 1),2) を同時に証明可能であること．
4) ソフトウェア領域および空き領域のサイズの多項式時間で検証が終了すること．

1.5 関連研究

ソフトウェア領域の改ざんを，特別なハードウェアを使用せずに行う方法として，Seshadri が提案した方式がある [9]．Seshadri らの方式では，検証機器と検証対象機器は擬似乱数生成関数を共有し，検証機器は challenge として

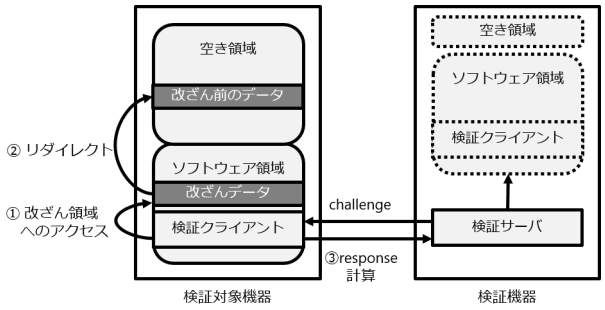

図 2 空き領域に攻撃ソフトウェア領域の改ざんしたい部分をコピーする攻撃

Fig. 2 The attacker copies the data to make legitimate response.

擬似乱数生成関数のシードを送信する．検証対象機器は擬似乱数生成関数が生成する乱数に対応するアドレスのデータから，連結ハッシュに似たチェックサムを計算する．

アドレスをランダムに指定することで，改ざんされていない機器が返す response をあらかじめ計算しておくことを困難にする．また，攻撃者がソフトウェア領域の改ざんしたい部分を空き領域にコピーしたのちに対象を改ざんし，チェックサムの計算にコピーを利用する攻撃（図 2 参照）に対しては，指定したアドレスを空き領域の対応するアドレスへリダイレクトするために，タイムラグが生じることを利用し，計算時間が長い場合は異常として検知する手法をとっている．

Seshadri らの方式はソフトウェアのみで機器の完全性を証明するため，特別なハードウェアを必要とせず，IoT 機器等の組込機器に広く適用できることが期待できる．一方で，タイムラグを利用する方法では，空き領域へのコピーを検知することはできるが，不正ソフトウェアが追加されていないことの証明はできていない．さらに，[10] では，Seshadri らの方式に対して，タイムラグをなくす攻撃の可能性も言及されている．空き領域が確実に空いていることを証明することができれば，不正ソフトウェアの追加を防ぐことができる上に，このような検証を無効にする攻撃コードが追加される可能性も下げることができる．

本稿では，Proof of Space を用いて空き領域が確実に空いていることを証明することで，特別なハードウェアを利用せずに外部から機器の完全性を検証する方法を検討する．

2. 準備：Proof of Space

2.1 Proof of Space とは

Proof of Space は Proof of Works の代替技術として，Dziembowski らによって提案された [11]．Proof of Works はスパムや DoS 攻撃対策として提案され，近年ではビットコインの基盤的技術となっている．しかし，Proof of Works は，クライアントに多くの CPU リソースを利用させることで攻撃時に負荷を要求するというその性質上，多くの電力を消費することを避けられず，ビットコインの普及により，その電力負荷が課題となっていた．その課題を解決するために，クライアントに対して負荷として CPU リソースを要求する代わりに，一定のメモリスペースの利用を要求する Proof of Space が Dziembowski らによって提案された [11]..

Dziembowski らが Proof of Space を提案した 2015 年以降，Proof of Space は改良が加えられているが，基本的なコンセプトとしては，特別な特徴を持つグラフ を生成し，各ノードにある値を対応させ，すべての値を計算するためには一定以上の領域の利用が必要であることをもって，すべての値を計算したことを証明することで Proof of Space を実現する．本セクションでは，本稿でベースとする Ren らが提案した Proof of Space on Stacked Expanders[12] の概要を説明する．

2.2 Graph Labeling

有向非巡回グラフ $G = (V, E)$ を考える．ここで，V は G に含まれるノードの集合，E はエッジの集合である．V に含まれるノードの個数を $|V| = N$ として，各ノードに 1 から N までの数値でラベリングをする．また，全てのノード $v \in V$ に対して，数値 $\omega(v) \in \{1,0\}^\lambda$ を対応させる．ここで，$V' = \{v_1, ..., v_n\}, v_i \subset V$ について，$\omega(V') = (\omega(v_1), ..., \omega(v_n))$，と定義する．また，$v$ の親ノードの集合を $\pi(v) = \{v' | (v', v) \in E\}$ と定義する．

このとき，各ノードに対応する数値を $\omega(v) = H(v, \omega(v, \pi(v)))$ と定義する．ここで，H はハッシュ関数 $H : \{0,1\}^* \to \{0,1\}^\lambda$ である．

Proof of Space では，全てのノードにおける $\omega(v)$ を計算するために一定以上のメモリ容量を要求するような G を構築することで，Proof of Space を実現する．

2.3 Bipartite Expanders

Bipartite Expanders は Proof of Space を実現するグラフの基本ブロックとなる．Bipartite Expanders とは以下の特徴を持った有向二部グラフである．

定義 1 $0 < \alpha < \beta < 1$ について， n 個のソースと n 個のシンクがあり，シンクの任意のサイズ αn のサブセットが，少なくとも βn 個のソースと接続している有向二部グラフを (n, α, β) *bipatite expander* と呼ぶ．ここで，出力エッジを持たないノードをシンク，入力エッジを持たないノードをソースを呼ぶ．

Bipartite Expanders は以下の定理をもって生成することができる．

定理 1 n 個のソース，シンクから，それぞれ d 個の出力エッジ，入力エッジを生成ことを想定する．各出力エッジに対応する入力エッジをランダムに選択し，接続する．このとき，d について以下の条件を満たすとき，このグラ

フは (n, α, β) *bipatite expander* となる．

$$d > \frac{H_b(\alpha) + H_b(\beta)}{H_b(\alpha) - \beta H_b(\alpha/\beta)} \quad (1)$$

ここで，$H_b(\alpha) = -\alpha \log_2 \alpha - (1-\alpha)\log_2(1-\alpha)$.

2.4 Localized Stacked Expanders

Stacked Expanders $G_{(n,k,\alpha,\beta)}$ は，$n(k+1)$ のノードを持ち，サイズ n のサブセット $V = \{V_0, V_1, ..., V_k\}$ で構成される．各 $i = 1, .., k$ について，V_{i-1} と V_i は，V_{i-1} をソース，V_i をシンクとした (n, α, β) *bipatite expander* となるようにエッジで接続されている．Stacked Expanders $G_{(n,k,\alpha,\beta)}$ に対して localize 処理を施したものを Localized Stacked Expanders $LG_{(n,k,\alpha,\beta)}$ と呼ぶ．Localize 処理とは，各 $i = 1, .., k$ について，$V_{i-1} = \{v_1, ..., v_n\}$ と $V_i = \{u_1, ..., u_n\}$ に対して，以下の処理をすることを言う．

1) 全ての i=1,...,n について，エッジ (v_i, u_i) を追加する．
2) 次に，全ての $i < j$ を満たすエッジ (v_i, u_j) をエッジ (u_i, u_j) に置き換える．

Localized Stacked Expanders の Graph Labeling について，以下の事実が証明されている．

定理 2 Localized Stacked Expanders $LG_{(n,k,\alpha,\beta)}$ において，V_k のサイズ αn のサブセットに含まれる全てのノードの ω を計算するとき，$\gamma n \lambda$ 以下のメモリ容量で計算をするには，$2^k \alpha n$ 回以上ハッシュ計算をする必要がある．ここで，$\gamma = \beta - 2\alpha$，λ はハッシュサイズである．

これより，Localized Stacked Expanders を用いることで全てのノードにおける $\omega(v)$ を計算するために一定以上のメモリ容量を要求するような G を構築できることがわかる．

2.5 ハッシュツリーを使った効率的な検証

多くの Proof of Space で採用されているハッシュツリーを使った効率的な検証方法を説明する．N 個のリーフのハッシュツリーを作ると，$\log N + 1$ 段のハッシュツリーとなる．ハッシュツリーについて，出力エッジの無い末端のノードを ϕ とおく．また，ϕ を 0 段目とし，以下 1 段目,...，$\log N$ 段目と呼ぶ．

定義 2 ノード v に到達可能なノードの集合を $\Pi(v)$，ノード v からの入力エッジを持つノードの集合を $\sigma(v) = v' (v, v') \in E$ とおく．$\log N$ 段目のノード c が選択されたとき，$i = 1, ..., \log N$ について，$v \notin \Pi(c)$ かつ $\sigma(v) \in \Pi(c)$ を満たす全ての v をかえす関数を $Open(c)$ と呼ぶ．

検証機器は検証対象機器に challenge C を送り，検証対象機器はすべての challenge $c \in C$ について，$Open(c)$ と $\pi(c)$ を検証機器に返す．検証機器はすべての $c \in C$ について，$H(\pi(c)) = c$ が成り立つこと，すべての $c \in C$ について，$H(Open(c)) = \phi$ が一致することを確認する．検証機器は上記が成り立つとき，検証対象機器が N 個のノードに対応する w を正しく計算したことを高い確率で保証することができる．

2.6 Proof of Space on Stacked Expanders[12]

本稿でベースとする Proof of Space on Stacked Expanders[12] は，Localized Stacked Expanders によって生成したグラフへの Graph Labeling と，ハッシュツリーを使った検証からなる．プロトコルのステップは以下の通りである．

1) 検証機器はノード数 $n(k+1)$ の Localized Stacked Expanders $LG_{(n,k,\alpha,\beta)}$ を生成し，検証対象機器に送信する．
2) 検証機器はさらに challenge C を送信する．
3) 検証対象機器は検証用にサイズ n のスペースを確保する．
4) 検証対象機器は受信した $LG_{(n,k,\alpha,\beta)}$ をもとに確保したスペースに V_1 を書き込む．
5) 検証対象機器は，さらに LG をもとに V_2 を計算し，$i = 1, ..., n$ について $v_i \in V_1$ が書き込まれた領域に $u_i \in V_2$ を上書きしていく．
6) 検証対象機器は，上記ステップを V_k まで繰り返す．
7) 検証対象機器は，上記計算と並行してすべての challenge $c \in C$ に対する $Open(c)$ と $\pi(c)$ を計算する．
8) 検証対象機器は，すべての challenge $c \in C$ に対する $Open(c)$ と $\pi(c)$ を送信する．
9) 検証機器は，すべての $c \in C$ について，$H(\pi(c)) = c$ が成り立つこと，すべての $c \in C$ について，$H(Open(c)) = \phi$ が一致することを確認する．

3. 提案方式

空き領域が確実に空いていることを証明するために最もシンプルな方法は，空き領域を乱数で埋め，その乱数が正しく格納されていることを確認する方法だろう．しかし，このシンプルな方式では，揮発性メモリについては証明ができない．本稿では，2.6 で説明した Proof of Space from Stacked Expander を用いて空き領域とソフトウェア領域の完全性を証明する手法を提案する．なお，シンプルな方式については付録にて考察をおこなう．

3.1 概要

本稿では，ソフトウェアの改ざん検知と Proof of Space を同時に行うため，サイズ n の空き領域上で $LG_{(n,k,\alpha,\beta)}$ の $V_1, ..., V_k$ を展開する一方で，$LG_{(n,k,\alpha,\beta)}$ を計算していく過程でランダムにソフトウェア領域のブロックを指定し，ソフトウェア領域が改ざんされていないかを検証する．

3.2 方式詳細

検証機器は検証対象機器の空き領域のサイズと，ソフト

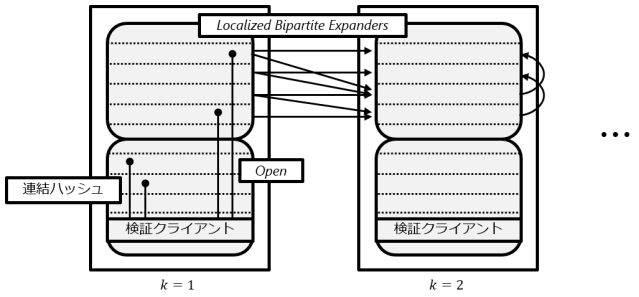

図 3　提案方式
Fig. 3　Proposed method

ウェア領域の内容を知っていることを前提とする．また，検証機器と検証対象機器は擬似乱数生成関数を共有していることを前提とする．ソフトウェア領域は，アドレス等によって任意のブロックに分割されており，ブロック x に格納されたデータを $d(x)$ とおく．

1) 検証機器はノード数 n の Localized Bipertite Expanders $LBG_{(n,\alpha,\beta)}$ を生成し，検証対象機器に送信する．
2) 検証機器はさらに challenge $C = \{C_1, c_2, l\}$ を送信する．ここで，C_1 は空き領域，c_2 はデータ領域に対する challenge である．
3) 検証対象機器は，検証用にサイズ n のスペースを確保する．
4) 検証対象機器は，受信した $LBG_{(n,\alpha,\beta)}$ をもとに確保したスペースに V_1 を書き込む．
5) 検証対象機器は，すべての challenge $c_1 \in C_1$ に対する $Open(c_1)$ と $\pi(c_1)$ を計算する．
6) 検証対象機器は，challenge c_2 を擬似乱数生成関数のシーズとして乱数列 $R(c_2) = \{r_1, ..., r_l\}$ を生成する．
7) 検証対象機器は，連結ハッシュ $z(c_2) = H(d(r_1), ..., d(r_l))$ を計算する．
8) 検証対象機器は，response $res = \{Open(c_1), \pi(c_1), z(c_2)\}$ を送信する．
9) 検証機器は，すべての $c \in C$ について，$H(\pi(c)) = c$ が成り立つこと，すべての $c \in C$ について，$H(Open(c)) = \phi$ が一致すること，連結ハッシュ $z(c_2) = H(d(r_1), ..., d(r_l))$ が検証機器が保持する情報から計算した値と一致するか否かを検証する．
10) 上記のやり取りを k 回繰り返す．

3.3　セキュリティ

ソフトウェア領域確認　提案方式では，kl 回ソフトウェア領域を探索する．このとき，サイズ m のソフトウェア領域のうち，ϵm が改ざんされていることを検知する確率は，$(1-\epsilon)^{kl}$ となる．$\epsilon = \frac{1}{kl}$ とおくと，kl が十分大きいとき，この確率は e^{-kl} となる．kl を十分大きくすれば，ソフトウェア領域の改ざんを見逃す確率は指数関数的に低くなる．このとき，検証対象機器と検証機器の計算量はともに kl である．

空き領域確認　[12] の議論を参考にすると，$\gamma = \beta - 2\alpha$ を 1 に十分近づければ，空きスペースに十分近い容量の領域 γn を利用したことを証明できるため，空き領域に不正なソフトウェアが追加されていないことを証明することができる．ただし，$\gamma = \beta - 2\alpha$ を 1 に近づけるにつれ，各ノードの入力エッジの数 d が指数関数的に大きくなることが知られている．LBG の計算量が dnk だから，証明できる領域のサイズは計算量とのトレードオフになる．[12] では，$0.7 < \gamma < 0.9$ を推奨している．

ソフトウェア領域確認と空き領域確認の同時性　ソフトウェア領域の改ざんまたは空き領域確認への不正ソフトウェアの追加のいずれかまたは両方発生していると，前述の確率でいずれかまたは両方で検知できるため，ソフトウェア領域の改ざんと空き領域の確認が同時にできることがわかる．

計算量　検証対象機器の計算量は $k(2n - 1 + l + dn)$ である．また，検証機器の計算量は検証対象機器の計算量は $k(|C_1| \log n + l)$ である．

4.　考察

提案手法はソフトウェアベースの完全性確認システムであり，検証対象機器に特別なセキュリティコンポーネントを要求しない．この性質によって，IoT 機器等の組込系機器を含むさまざまな機器への適用が期待できる．

提案手法は，空き領域を乱数で埋める方式と異なり，一時データとして十分大きなデータを生成させ，生成したことを証明することで成り立っているため，検証開始時や検証終了後に検証用のデータを保持している必要はない．そのため，不揮発性領域に格納されたデータを不揮発性領域よりもサイズの大きい揮発性領域に展開する場合への適用が期待できる．

一方で，提案手法は，高い検知性能を求める場合，検証対象機器に膨大な計算量を要求する．そのため，計算リソースが小さい機器では検証に時間がかかることが予想される．計算の大部分が Proof of Space によるものであるため，Proof of Space のアルゴリズムの改善が今後の課題となる．

5.　結論

本稿では，特別なセキュリティコンポーネントを利用せず，ソフトウェアベースで完全性確認をおこなうシステムを提案した．提案手法は．Proof of Works の代替として提案された Proof of Space をベースとしている．提案手法は，確率的に改ざんを検知するシステムであり，検知率と計算量がトレードオフの関係となっているため，さらなる

効率化が今後の課題となる．

謝辞 本研究の遂行にあたり，アドバイスをいただいた富田潤一さんに深く感謝いたします．

参考文献

[1] Iot セキュリティガイドライン. https://www.meti.go.jp/press/2016/07/20160705002/20160705002.html. (Accessed on 08/08/2019).
[2] Github - jgamblin/mirai-source-code: Leaked mirai source code for research/ioc development purposes. https://github.com/jgamblin/Mirai-Source-Code. (Accessed on 08/01/2019).
[3] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, Vol. 2015, p. 91, 2015.
[4] Richard Wilkins and Brian Richardson. Uefi secure boot in modern computer security solutions. In *UEFI Forum*, 2013.
[5] Intel Corporation. Intel trusted execution technology: Software development guide. 2017.
[6] Intel Corporation. Intel software guard extensions developer guide. 2017.
[7] GlobalPlatform Technology. Root of trust definitions and requirements version 1.1. 2018.
[8] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, Vol. 2004, 2004.
[9] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pp. 272–282. IEEE, 2004.
[10] Markus Jakobsson. Secure remote attestation. *IACR Cryptology ePrint Archive*, Vol. 2018, p. 31, 2018.
[11] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Annual Cryptology Conference*, pp. 585–605. Springer, 2015.
[12] Ling Ren and Srinivas Devadas. Proof of space from stacked expanders. In *Theory of Cryptography Conference*, pp. 262–285. Springer, 2016.

付　　録

A.1 空き領域を乱数で埋める方式

本セクションでは空き領域を乱数で埋めるについて考察する．

A.1.1 方式詳細

検証対象機器の空き領域は全て乱数で埋められており，検証機器は空き領域の内容とソフトウェア領域の内容を知っていることを前提とする．このとき，[9] によく似た検証方式が考えられる

1) 検証機器と検証対象機器はあらかじめ擬似乱数生成関数を共有しておく．
2) 検証機器はランダムに challengec を送信する．
3) 検証対象機器は challengec を擬似乱数生成関数にシードとして入力し，擬似乱数列を生成する．
4) 検証対象機器は，乱数に対応するブロックの連結ハッシュを計算し，検証機器に送信する．
5) 検証機器は，検証対象機器と同様に連結ハッシュを計算し，response と一致するか否かを検証する．

A.1.2 提案手法との比較

この方法では，検証対象機器は，乱数を保持している必要があるため，揮発性領域には適用できない．

サイズ $\epsilon(n+m)$ が改ざんされたことを検知できない確率は $(1-\epsilon)^l$ で，$\epsilon = 1/l$ とおくと，l が十分大きい時，その確率は e^{-l} となる．また，計算量は検証対象機器，検証機器ともに l となる．

すなわち，ソフトウェアをインストール後，空き領域への書き込み等がなく，空き領域のサイズが不変である不揮発性領域の完全性確認をする場合には，この方法は，提案手法よりも少ない計算量で改ざんを検知することができる．