Intel SGX を利用した自己破壊・出現データの実装

加藤 風芽[1,a)]　新城 靖[1,b)]

概要：今日ではインターネットを介してより多くのセンシティブな情報がやりとりされるようになってきており、これらの情報を安全に保つことが求められる。機密実行環境は、プログラムを他者の計算機でコードを秘匿したまま実行することを可能にする。本研究では、Intel SGX を利用して機密実行環境を構築し、指定した時刻になると自動的に使用できなくなるデータ（自己破壊データ）および、逆に指定した時刻になると使用できるようになるデータ（自己出現データ）を実現する。自己破壊・出現データを出力する条件は、S 式に基づくドメイン固有言語で記述できる。自己破壊・出現データを実現するために、本研究では汎用サービスとして提供可能な外部の信頼された時刻源およびモノトニックカウンタを利用する。提案方式は、ファイルに保存された状態に対するロールバック攻撃や、ネットワーク経由で送受信されるメッセージのリプレイ攻撃に耐性がある。実験の結果、自己破壊・出現データの転送および出力が実用的な時間で行えることを確認した。

キーワード：自己破壊データ, 自己出現データ, Intel SGX, 機密実行環境

1. はじめに

私たちがインターネットを介して日常的にやりとりするデータの中には、個人情報などのセンシティブなデータが多く含まれる。また、電子書籍や遠隔授業における試験問題など、データの公開期間を制限したいケースが多くある。

自己破壊データ (self-destructing data) は、時間経過などの条件が満たされた時、自動的に利用できなくなる仕組みを持ったデータである [1]。例えば、センシティブな情報を自己破壊データの形式で受け渡すようにすれば、ユーザおよびサービス提供者は情報の漏洩や不適切な利用のリスクを避けられる。Instagram や Facebook などの SNS (Social Networking Service) におけるストーリー機能（24 時間で消える投稿）も、自己破壊データの一種といえる。

しかし、従来の自己破壊データの実装では、次のような問題点がある。

- 信頼できる中央サーバ、または、大規模な DHT (Distributed Hash Table) を必要とする。
- プログラムの改変による不正な出力を防ぐことができない。

そこで本研究では、機密実行環境 (Trusted Execution Environment) を用いて自己破壊データを実装することを提案する。機密実行環境とは、CPU で実装された、プログラムのデータを他のプログラムから秘匿したまま実行することができる環境のことである。本研究では具体的に Intel CPU の拡張命令である Intel SGX (Software Guard Extensions) を利用して自己破壊データを実装する。Intel SGX の enclave （飛び地）と呼ばれる機密実行環境では、実行されるプログラムのメモリ空間が他のプログラムからハードウェアのレベルで保護される。本研究では、これにより、中央サーバや DHT を用いずに個人の計算機だけで完結する自己破壊データを実現する。また、自己破壊とは逆に、未来のある時刻に利用可能になる**自己出現データ** (self-emerging data) [2] も実現する。

自己破壊・出現データの応用として、次のようなことが挙げられる。

- 通信販売で、商品の配達中のみアクセス可能な形で住所を保持する。
- SNS で、時間経過後にメッセージへのアクセスが不能になるストーリー機能を実現する。
- 出版物の著作権保護のため、アクセス回数を制限する。
- オンライン試験の問題を、試験時間中のみアクセス可能にする。

2. 自己破壊・出現データ

2.1 用語の定義

本論文で利用する用語を、以下のように定義する（**図 1**）。

[1] 筑波大学
University of Tsukuba
[a)] fkato@softlab.cs.tsukuba.ac.jp
[b)] yas@cs.tsukuba.ac.jp

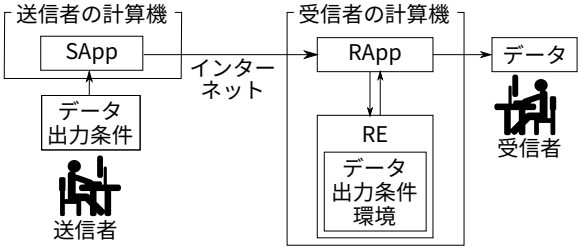

図 1 本論文で利用する用語

自己破壊・出現データ

自己破壊データ (self-destructing data) とは、ある条件を満たすと、以降利用できなくなるデータのことである。自己出現データ (self-emerging data) は、逆に、ある条件を満たして初めて利用できるデータである。本研究では、どちらも同様に扱う。

出力条件

出力条件とは、自己破壊・出現データの出力を許可する条件のことである。本研究では、DSL （ドメイン固有言語）により出力条件を記述可能にする。

環境

環境とは、出力条件中に出現する変数とその変数に束縛された値との対応関係のことである。

送信者

送信者 (sender) は、自らが保持するデータを自己破壊・出現データの形式で送信する主体である。送信者は、インターネットに接続された計算機でプログラムを実行する。送信者が実行するプログラムを、SApp (sender application) と呼ぶ（図 1)。

受信者

受信者 (receiver) は、送信者から自己破壊・出現データを受け取り、これにアクセスする主体である。受信者も送信者と同様、インターネットに接続された計算機でプログラムを実行する。受信者が実行するプログラムは、SGX の enclave と、enclave 外の通常のプログラムからなる。前者を RE (receiver enclave)、後者を RApp (receiver application) と呼ぶ (図 1)。データを受け取る際に、RApp はインターネットを経由して SApp と接続される。

RApp は、受信者からデータの出力を求められた場合、RE に要求を送る。ここで出力条件を満たしたときのみ、RE はデータを出力する。この時、SApp は不要である。

2.2 目標

本研究では、次のような特徴をもった自己破壊・自己出現データを実現する。

- 柔軟な出力条件を指定できる。具体的には、時刻の範囲、アクセス回数および、それらを組み合わせた論理式が利用できる。
- 個人の計算機で完結可能である。信頼できる中央サーバや大規模な DHT を必要としない。
- 耐タンパデバイスが利用可能であれば、オフラインでデータを開封できる。

表 1 出力条件で用いるデータ型

型	説明
I64	64-bit 符号付き整数
Bool	ブール値
String	文字列
α	任意型（型変数）
[α]	α のリスト

表 2 出力条件で用いる組込み関数

組込み関数	型	説明
`<`	I64 → I64 → Bool	小なり
`>`	I64 → I64 → Bool	大なり
`==`	$\alpha \to \alpha \to$ Bool	等しい
`and`	[Bool] → Bool	AND
`or`	[Bool] → Bool	OR
`not`	Bool → Bool	NOT
`timevalue`	String → I64	時刻をパース
`now`	I64	問合せた現在時刻
`++`	I64	変数の値の取得とインクリメント

2.3 出力条件 DSL

出力条件を表現するための DSL として、S 式を利用する。用意する型を**表 1**、組込み関数を**表 2** に示す。ここで、時刻は Unix Epoch からの経過時間（現在のところミリ秒単位）として 64-bit 符号付き整数で表現する。

以下に実際の出力条件の例を示す。

自己破壊データの例。2020 年 12 月 1 日以降、アクセス不能になる。

```
(< (now)
   (timevalue
     2020-12-01T00:00:00.0000Z))
```

自己出現データの例。2020 年 12 月 1 日以降にアクセス可能になる。

```
(> (now)
   (timevalue
     2020-12-01T00:00:00.0000Z))
```

複雑な自己破壊データの例。2020 年 12 月 1 日以降かつ最大 10 回までアクセス可能。

```
(and (> (now)
        (timevalue
          2020-12-01T00:00:00.0000Z))
     (< (++ x) 10))
```

3. 設計

3.1 利用するハードウェアと外部のサービス

本研究で自己破壊・出現データを実現するために利用するハードウェアと外部のサービスについて述べる。

3.1.1 機密実行環境

機密実行環境とは、CPU で実装された、プログラムのデータを他のプログラムから秘匿したまま実行することができる環境のことである。具体例としては、Intel の Core CPU に搭載されている Intel SGX (Software Guard Extensions) や ARM に搭載されている TrustZone などがある。これらは多くの PC やスマートフォンで利用することができ、決済情報の保護や著作権管理のために利用され始めている [3]。

本研究では、機密実行環境の実装として Intel SGX[4] を利用する。SGX は Intel の第 6 世代 Core CPU (2015 年) 以降から搭載されている拡張命令で、enclave とよばれる機密実行環境を提供する。SGX では、 enclave と通常のプログラムの間の API は相互の関数呼び出しとなっており、enclave への呼び出しは ECall (Enclave Call)、enclave 外への呼び出しは OCall (Outside Call) と呼ばれる。

3.1.2 Remote Attestation と鍵共有

機密実行環境で実行されるコードが真正であることを別の計算機から検証する仕組みを、Remote Attestation (RA) とよぶ。SGX は認証つき楕円曲線 Diffie-Hellman 鍵共有を応用した RA を提供しており、RA の後、遠隔のプロセスと enclave との間に共通鍵によるセキュアチャネルが構築できる。本研究ではこの仕組みを使い、SApp から RE へ自己破壊・自己出現データを転送する。

3.1.3 時刻源

出力条件に時刻が含まれる場合、信頼できる時刻データが必要になる。計算機で時刻を得る通常の方法は信頼できないため、署名された時刻データを提供するサーバまたは耐タンパデバイスを用意する。これらを TTP (Trusted Time Provider) と呼ぶ。TTP は乱数 (nonce) を受け取り、それを時刻とともに署名して送り返す。この署名及び nonce の一致を検証することで、時刻が本物であり、かつリプレイではないことを確かめられる。ここで、TTP は証明書によって認証されていればよく、汎用サービスとして送信者とは別の主体が用意しても構わない。

この方式は、RFC 3161[5] で示されている Time-Stamp Protocol (TSP) と類似している。TSP は、あるデータがある時点で既に存在していたことを証明するためのプロトコルであり、タイムスタンプサーバにデータのハッシュを送ると、そのハッシュを時刻とともに署名したデータが返される。この署名されたデータをあとで検証すれば、データの存在していた時刻を確かめられる仕組みになっている。本研究の時刻源はデータのタイムスタンプが目的ではないため、単に nonce を使うようになっている。

3.1.4 モノトニックカウンタ

永続データに対するロールバック攻撃を防ぐために、モノトニックカウンタを利用する。これは、値が増加することが保証されているカウンタである。この実装として、時刻源と同様にサーバまたは耐タンパデバイスを用いる。これらのことは MCP (Monotonic Counter Provider) と呼ぶ。MCP も TTP と同様に証明書により認証されていればよく、汎用サービスとして送信者とは別の主体が用意してもかまわない。モノトニックカウンタを実装する手法としては、ROTE[6] のように、複数の機密実行環境で動作するプロセス群を協調させる手法も考えられる。

3.1.5 永続化のためのファイルシステム

Enclave 内では、その enclave に固有の鍵 (seal 鍵) を利用してデータを暗号化することができる。この仕組みを data sealing[7] と呼ぶ。本研究ではこの仕組みを使い、seal されたデータをファイルシステムに保存する。ファイルシステムとしてローカル OS のものを利用し、また、TTP や MCP を耐タンパデバイスで実装すれば、オフラインでのデータの出力が可能になる。

3.2 脅威モデル

送信者側の要素（SApp とそれを実行する計算機）と、TTP、および MCP は安全であると仮定する。想定される受信者は malicious であり、以下のような攻撃を行う。

ロールバック攻撃

Enclave を実行する前に、暗号化されたファイルをバックアップする。Enclave の実行後、更新されたファイルを破棄し、バックアップを復元することにより、以前の状態を復元しようとする。

リプレイ攻撃

TTP や MCP から送られてくるデータをキャプチャし、後で enclave に渡すことで、時刻やカウンタの値をごまかそうとする。

コードの改ざん

Enclave のコードを改ざんし、出力条件を満たしていなくてもデータを出力させるようにする。

受信者は、自己破壊・出現データから出力されたデータをそのまま保存しようとするかもしれない。そのような攻撃を防ぐには、保護されたデバイスやハイパバイザが必要になる [8][9]。本研究では、出力されたデータを保護するにはこのような既存の手法を用いることとし、研究の範囲外とする。

3.3 データの転送

データの転送手順を**図 2** に示す。この図では、送信者から受信者へデータ M および出力条件 C を送信する。図中

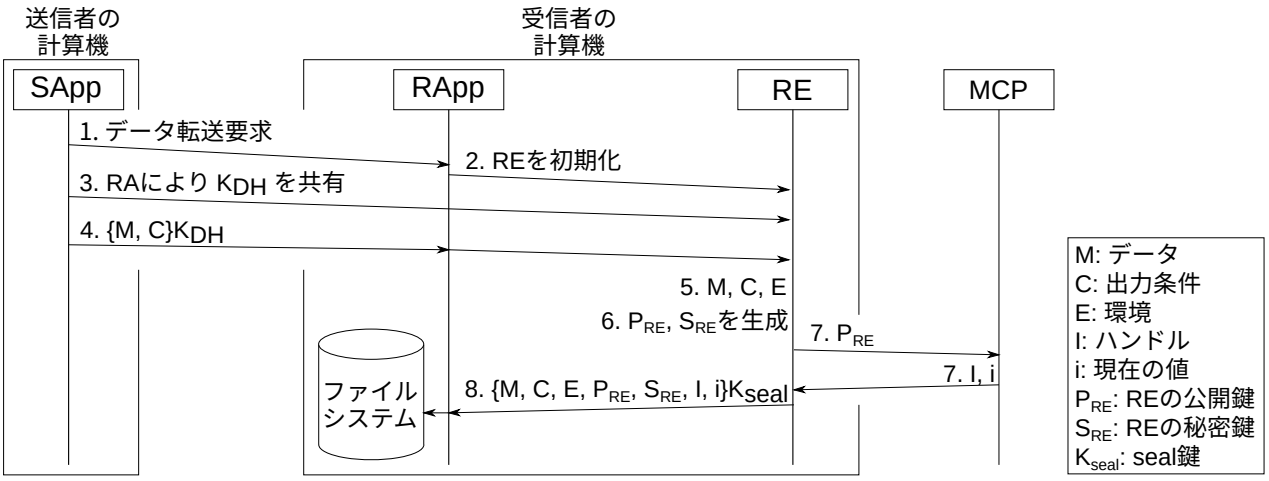

図 2 データ転送手順

の SApp と RApp は、enclave の外で動作する通常のプログラムである。順を追って説明する。

(1) SApp が RApp にデータ転送の準備を要求する。
(2) RApp は RE を初期化する。
(3) SApp が RE に対し RA を行う。ここで送信者と RE が共通鍵 K_{DH} を得る。K_{DH} は RE 内に秘匿され、RApp からは取り出せない。
(4) SApp は M と C を連結したデータ (M,C) を K_{DH} で暗号化した暗号文 $\{M,C\}_{K_{DH}}$ を RApp 経由で RE に送る。
(5) RE はこの暗号文を復号し、M,C を得る。C に出現する変数を列挙し、環境 E を初期化する。
(6) RE は公開鍵暗号の鍵ペアを生成する。公開鍵を P_{RE}、秘密鍵を S_{RE} とする。
(7) RE は RApp 経由で MCP に P_{RE} を含んだカウンタ作成要求を送り、カウンタのハンドル I と初期値 i を得る。
(8) RE は $M, C, E, P_{RE}, S_{RE}, I$ および i を seal 鍵 K_{seal} で暗号化し、RApp を経由して永続化のためのファイルシステムに保存する。

3.4 データの出力

データの出力手順を**図 3** に示す。この図では、ユーザが RApp を使い、ファイルシステムに保存された暗号化されたデータを機密実行環境 RE に渡し、出力を要求する。順を追って説明する。

(1) RApp は RE を初期化する。RE は、ファイルシステムに保存されたデータを読み出す。 RE は seal 鍵 K_{seal} でデータを復号し、データ M、出力条件 C、環境 E、モノトニックカウンタ用の秘密鍵 S_{RE} と公開鍵 S_{RE}、モノトニックカウンタのハンドル I、モノトニックカウンタの値 i を得る。この保存されているカウンタ値を i_{stored} と呼ぶ。
(2) RE は I と S_{RE} を使い、MCP から現在のカウンタ値 i を得る。署名と nonce を検証し、不正であればエラーを出力して終了する。次に、i_{stored} と i を比較し、異なっていればロールバック攻撃としてエラーを出力して終了する。
(3) RE は出力条件 C を評価する。C の中に時刻があれば、TTP から時刻を得る。署名と nonce を検証し、不正であればエラーを出力して終了する。
(4) C に++など、副作用のある式が含まれる場合、変数の値が変化し、E が書き換えられる。新しい環境を E' と呼ぶ。
(5) RE は評価の結果 C が満たされなければエラーを出力して終了する。
(6) RE は I と S_{RE} を使い、モノトニックカウンタの値 i をインクリメントする。得られる新しいカウンタ値を i' と呼ぶ。
(7) RE は M, C, E', P_{RE}, S_{RE}, I および i' を seal 鍵 K_{seal} で暗号化し、ファイルに保存する。
(8) RE はデータ M を出力する。

3.3 節で述べたデータの転送とは異なり、データの出力においては、RA を行わない。RE が改ざんされていた場合は、K_{seal} を取り出すことができず、データの出力ができないため、受信者は RE を改ざんする意味がないからである。受信者は、RE が不正な動作をしないことを確認するために local attestation を実行したり、不正な動作による被害を最小限にするためにコンテナ等で隔離された環境で RApp と RE を動作させることができる。

3.5 TTP へのアクセス

RE から TTP へのアクセスは、**図 4** に示す手順で行う。ここで、図中の「()」は署名を検証されないメッセージ、「{}」は署名を検証されるメッセージを表す。また、図中では RE と TTP を直接接続しているが、厳密にはネット

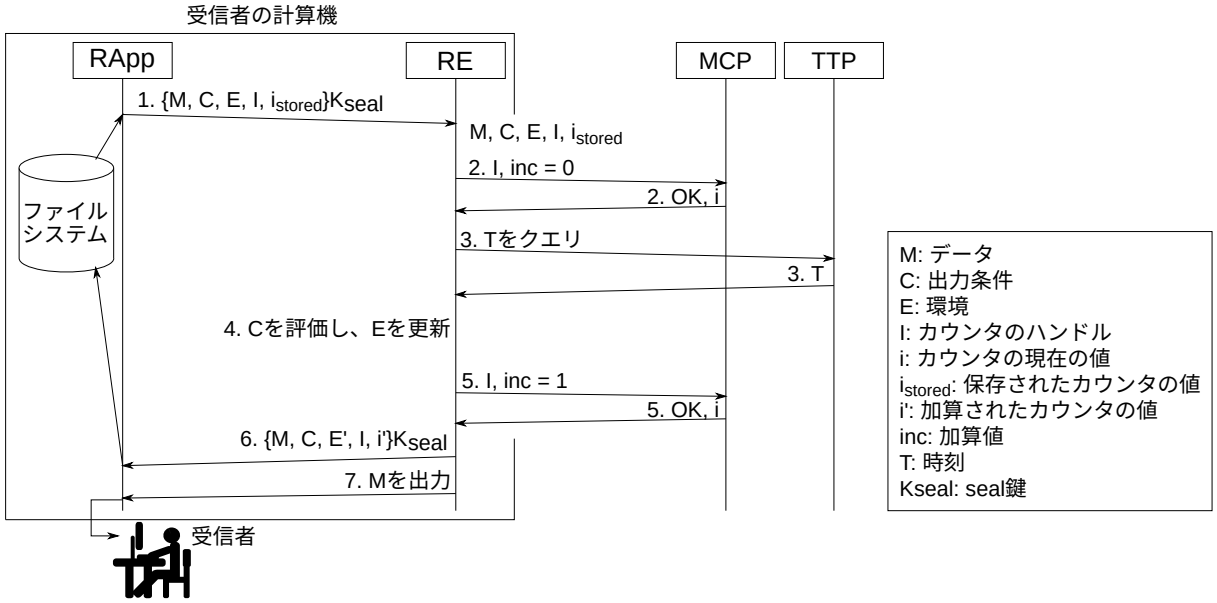

図 3 データ出力手順

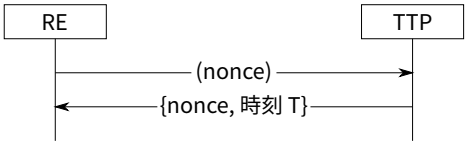

図 4 TTP へのアクセス手順

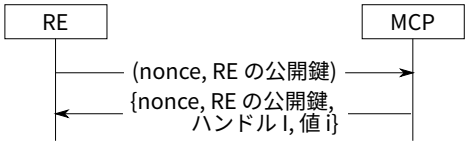

図 5 モノトニックカウンタの初期化手順

表 3 TTP へのアクセスで交換されるメッセージの例

方向	ペイロードの内容
RE → TTP	`{"msgtype": "time_query", "nonce": 1234}`
TTP → RE	`{"msgtype": "time_answer", "nonce": 1234, "time": 1601286372.123}`

ワークアクセスは RApp への OCall を介して行われる。順を追って説明する。

(1) RE は nonce として使う乱数を生成する。
(2) RE は nonce を TTP へ送信する。
(3) TTP は nonce と時刻の値を署名して RE に返す。
(4) RE は受け取ったメッセージの署名を検証する。検証できなければ、エラーとして処理する。
(5) RE は 送った nonce と受け取った nonce が一致していることを確認する。一致していなければ、リプレイ攻撃がなされたとして、エラーとして処理する。
(6) ここまでエラーがなければ、RE は時刻を信頼できるものとして受理する。

表 3 に、このプロトコルに沿ったメッセージのやり取りの例を示す。この表でペイロードは、JSON で記述されている。この例では、RE は、`"time_query"`という要求を nonce 1234 と共に TTP に送っている。TTP は、nonce 1234 と時刻を送り返している。

3.6 MCP へのアクセス

3.6.1 モノトニックカウンタの初期化

モノトニックカウンタの初期化は、**図 5** に示す手順で行う。図 5 においても、3.5 節と同様、「()」は署名を検証されないメッセージ、「{}」は署名を検証されるメッセージを表す。ここで初期化とは、新たなカウンタを生成することである。順を追って説明する。

(1) RE は nonce として使う乱数を生成する。
(2) RE は nonce と RSA 公開鍵を MCP に送信する。
(3) MCP はカウンタの領域を確保する。領域に対応するハンドル I がランダムに割り当てられる。
(4) MCP はカウンタの初期値 i を設定する。
(5) MCP は確保した領域に、I をキーとして i と受け取った公開鍵を保存する。
(6) MCP は nonce と 受け取った公開鍵、I および i を署名して RE に返す。
(7) RE は受け取ったメッセージの署名を検証する。検証できなければ、エラーとして処理する。
(8) RE は送った nonce と受け取った nonce が一致していることを確認する。一致していなければ、エラーとして処理する。リプレイ攻撃は、ここで検出される。
(9) RE は送った公開鍵と受け取った公開鍵が一致していることを確認する。一致していなければ、エラーとして処理する。
(10) ここまでエラーがなければ、RE は受け取った I およ

表 4 モノトニックカウンタの初期化で交換されるメッセージの例

方向	ペイロードの内容
RE → MCP	{"msgtype": "ctr_init", "nonce": 1235, "pubkey": {"e":"AQAB", "kid":"xxxx", "kty":"RSA", "n":"XXXX"}}
MCP → RE	{"msgtype": "ctr_init_ok", "nonce": 1235, "pubkey": {"e":"AQAB", "kid":"xxxx", "kty":"RSA", "n":"XXXX"}, "handle": 3456, "ctr": 42}

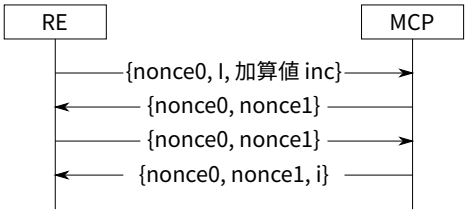

図 6 モノトニックカウンタへのアクセス手順

び i の値を利用する。

表 4 に、このプロトコルに沿ったメッセージのやり取りの例を示す。この例では、RE は、"ctr_init"という要求を nonce 1235 および自身の公開鍵と共に MCP に送っている。MCP は、nonce 1235 および RE の公開鍵とともに、カウンタのハンドル 3456 と初期値 42 を送り返している。

3.6.2 モノトニックカウンタへのアクセス

初期化済みのモノトニックカウンタへのアクセスは、**図 6** に示す手順で行う。順を追って説明する。ここで、リプレイを防ぐため、クライアントとサーバの両方で nonce を生成する。

(1) RE は、カウンタに対して加算する値 inc を設定する。カウンタ値を得たい場合は 0、インクリメントしたい場合は 1 が設定される。
(2) RE は nonce0 として使う乱数を生成する。
(3) RE は nonce0 と カウンタのハンドル I および inc を RE の秘密鍵で署名して MCP に送信する。
(4) MCP は受け取ったメッセージの署名を I の示す領域に保存された公開鍵で検証する。検証できなければ、RE にエラーメッセージを返して通信を終了する。
(5) MCP は nonce1 として使う乱数を生成する。
(6) MCP は nonce0 と nonce1 を署名して RE に返す。
(7) RE は受け取ったメッセージの署名を検証し、nonce0 が送ったものと一致することを確認する。
(8) RE は nonce0 と nonce1 を署名して MCP に返す。
(9) MCP は受け取ったメッセージの署名を検証し、nonce1 が送ったものと一致することを確認する。

表 5 モノトニックカウンタへのアクセスで交換されるメッセージの例

方向	ペイロードの内容
RE → MCP	{"msgtype": "ctr_access", "nonce0": 2222, "handle": 3456, "inc": 1}
MCP → RE	{"msgtype": "ctr_access_ack0", "nonce0": 2222, "nonce1": 3333}
RE → MCP	{"msgtype": "ctr_access_ack1", "nonce0": 2222, "nonce1": 3333}
MCP → RE	{"msgtype": "ctr_access_ok", "nonce0": 2222, "nonce1": 3333, "ctr": 43}

(10) MCP はカウンタの値に inc だけ加算する。
(11) MCP は nonce0 と nonce1 および カウンタの新しい値 i を署名して RE に返す。
(12) RE は受け取ったメッセージの署名を検証し、nonce0 と nonce1 が正しいことを確認する。
(13) ここまでエラーがなければ、RE は新しいカウンタの値を利用する。

表 5 に、このプロトコルに沿ったメッセージのやり取りの例を示す。この例では、RE は、"ctr_access"という要求を nonce0 2222、ハンドル 3456、加算値 1 と共に MCP に送っている。MCP は nonce0 2222 と共に nonce1 3333 を送り返している。RE はさらに nonce0 2222 と nonce1 3333 を送り返している。最後に MCP は nonce0 2222、nonce1 3333 と共に、カウンタの新しい値 43 を送り返している。

4. 実装

4.1 SDK

本研究では、自己破壊・出現データを Intel SGX SDK および Rust SGX SDK を用いて実装している。前者は Intel 社が提供し、C/C++での enclave および通常環境のコードの開発を可能にする。また、EDL (Enclave Definition Language) とよばれる言語による関数プロトタイプの定義から、スタブを自動で生成する。後者は Rust 言語で enclave および通常環境のコードの開発を可能にするもので、Rust の標準ライブラリ (std) を enclave 内部から利用可能にする。標準ライブラリの機能のうち、標準入出力、ネットワーク、ファイルシステムなど、外部へのアクセスが必要なものに関しては、OS の API をラップした手続きを OCall を介して利用している。

4.2 Intel Attestation Service を利用した Remote Attestation による鍵交換

本研究では、Intel 社が提供するオンラインサービスで

ある Intel Attestation Service (IAS) を利用して remote attestation を実現している。Intel SGX における remote attestation は、enclave の identity が求めているものと一致していることを別の CPU から検証することである。ここで enclave の identity としては、enclave の起動ログの SHA-256 ハッシュである MRENCLAVE と、enclave の署名者の RSA 公開鍵の SHA-256 ハッシュである MRSIGNER がある。MR は Measurement Register の略である。起動ログには enclave のページの全内容が含まれるので、MRENCLAVE が同一ならコードの内容も同一である。

Intel SGX では、remote attestation の前段階として local attestation を行う。Intel SGX における local attestation は、同一 CPU 上で動作している enclave の identity を、別の enclave から検証することである。このために、SGX には EREPORT および EGETKEY という命令がある。ある enclave A が、enclave B の identity を検証したい場合、まず enclave A は自らの MRENCLAVE を enclave B に渡す。次に enclave B がこの MRENCLAVE を引数に EREPORT 命令を実行すると、enclave B の identity やその他の情報と、これらに対する MAC（メッセージ認証コード）からなる REPORT と呼ばれる構造体が出力される。この MAC を生成するための秘密鍵は report 鍵と呼ばれ、EREPORT 命令と enclave A からしか見えない。Enclave B はこの REPORT を enclave A に渡す。Enclave A は EGETKEY 命令を実行して report key を得、REPORT の MAC を検証する。MAC が正しく、REPORT に含まれている enclave B の identity も正しければ、local attestation は成功である。

3.3 節で述べたデータの転送処理において、SApp と RE の間で行う Remote Attestation の方式としては、EPID-based Remote Attestation を利用した。EPID (Enhanced privacy ID) はグループ署名方式の一種である。グループ署名方式では、あるグループに多数の異なる秘密鍵を紐付けることができ、同一のグループに属する秘密鍵で署名されたメッセージは、単一のグループ公開鍵で検証することができる。Intel は CPU の種類ごとに EPID グループを割り当てている。CPU に紐付いた EPID の秘密鍵には、Quoting Enclave (QE) とよばれる特殊な enclave でしかアクセスできない。

本研究での remote attestation の手順は、次のとおりである。SApp は、公開鍵暗号の鍵ペアを生成し、公開鍵を RApp に送る。RApp は、 RE に QE の identity と受取った公開鍵を渡す。RE は、公開鍵暗号の鍵ペアを生成し、秘密鍵と SApp の公開鍵から K_{DH} を計算する。RE は QE に向けた REPORT を生成し、RApp に渡す。ここで、RE は REPORT に SApp の公開鍵と自分の公開鍵を追加データとして含める。RApp は QE に REPORT を渡す。QE は REPORT の検証に成功すると、REPORT の MAC を EPID 秘密鍵による署名に差し替えた QUOTE という構造体を出力する。RApp は QUOTE を SApp に送り返す。SApp がこの QUOTE の EPID 署名を検証できれば、remote attestation が完了する。この署名の検証処理は IAS に依頼する。最後に、SApp は QUOTE から取り出した RE の公開鍵と自らの秘密鍵から K_{DH} を計算する。

Intel SGX では IAS に依存しない ECDSA-based Remote Attestation という方式も提供され始めており、対応は今後の課題である。

4.3 RE の手続き

受信者の enclave (RE) を、Rust SGX SDK を用いて実装した。RApp は、**表 6** に示した RE の手続きを ECall として利用できる。`set_qe_info()`と `start_request()`は、remote attestation のための手続きである。`store_file()` は、自己破壊・出現データを受信し、ファイルシステムに保存するための手続きである。`output_file()`は、自己破壊・出現データの出力を行うための手続きである。

4.4 TTP および MCP

TTP と MCP のサーバを、Python 言語で実装した。RE から TTP および MCP への通信路には TCP を、データを署名する方式としては JWT (JSON Web Token)[10] を用いた。JWT は、Base64URL 方式でエンコードされたヘッダ、ペイロード、およびこれらの署名をピリオドで区切ったデータである。ヘッダおよびペイロードはデコードすると JSON 文字列になっている。ヘッダには署名アルゴリズムの情報が記述され、ペイロードとしては任意の JSON 文字列が使える。本研究の実装では、ペイロード中に `msgtype` というキーを用意し、これによってメッセージの種類を区別している。署名アルゴリズムとしては RSA-SHA256 を使っている。また、JWT を改行文字で区切ってメッセージ境界を表現している。

5. 評価

5.1 セキュリティ解析

3.2 節で示した脅威に対し、提案手法が防御できることを示す。出力条件として利用回数が指定されていた場合、受信者はロールバック攻撃を行い、ファイルシステムを以前の状態に戻すことで、環境ごと変数を古い値に戻し、本来の利用回数を超えてデータを出力させようとするかもしれない。この場合、外部のモノトニックカウンタからの値とファイルシステムに保存されたモノトニックカウンタの値の違いから、この攻撃を検出できる。受信者が時刻サーバからのメッセージのリプレイを行い、時刻をごまかそうとした場合、デジタル署名は検証に成功するが、nonce の値が一致しないので、この攻撃を検出できる。受信者がデー

表 6 RE が提供する ECall

ECall	目的

`set_qe_info(`
`  [in] sgx_target_info_t *target_info,`
`  [in] sgx_epid_group_id_t * epid_group_id)`
	RA のために QE の identity を設定する。

`start_request(`
`  [in] sgx_ec256_public_t *ga,`
`  [out] sgx_report_t *report)`
	RA および DH 鍵共有 のために SApp から送られてきた公開鍵を受取り、QE に向けた REPORT を生成する。

`store_file(`
`  [in] uint8_t *ciphertext,`
`  [in] uint64_t ciphertext_len,`
`  [in] const char *filename)`
	`ciphertext` と `ciphertext_len` で指定された送信者から受取った暗号文を復号し、seal 鍵で再び暗号化した後、`filename` で指定されたファイルに保存する。

`output_file(`
`  [in] const char *filename,`
`  [in] int64_t fd)`
	`filename` で指定されたファイルに保存された自己破壊・出現データの出力要求を行う。出力は、ファイル記述子 `fd` で指定されたファイルになされる。

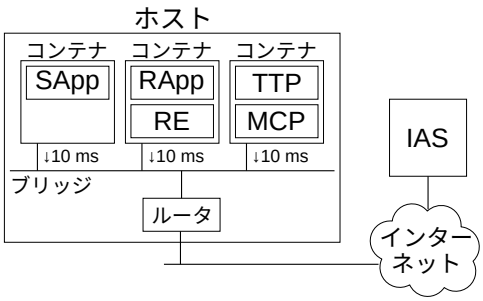

図 7 実験に用いた環境

タ転送の前に enclave のコードを改ざんした場合、RA が失敗するので、データを受け取ることはできない。また、データ転送後、データ出力前に enclave のコードを改ざんした場合、seal 鍵が変わってしまう。したがって、元の seal 鍵で暗号化された、ファイルシステムに保存されたデータを復号することができない。

5.2 実行時間

図 7 に示したハードウェアおよびネットワークを用いて、データの転送および出力の所要時間を測定した。実験に用いた計算機には、Intel Core i7-9700K CPU @ 3.60 GHz (8 cores) と 64 GB RAM が備わっている。実験には Docker を用い、送信者、受信者および TTP と MCP にそれぞれコンテナを用意した。各コンテナを Linux bridge に接続した。Linux bridge をホスト OS の IP 層をルータとして LAN に接続した。LAN は、学内ネットワークと SINET を経由してインターネットに接続されている。SApp は、IAS (Intel Attestation Service) とインターネット経由で通信する。ホスト OS は、Ubuntu 18.04 で Linux kernel 5.4.0 を実行している。現実のインターネットにおける通信遅延をエミュレートするため、以下のように Linux の tc コマンドを用い、各コンテナでパケットを送信する際 10 ms 遅延が入るようにした。

```
tc qdisc add dev eth0 root netem \
delay 10ms
```

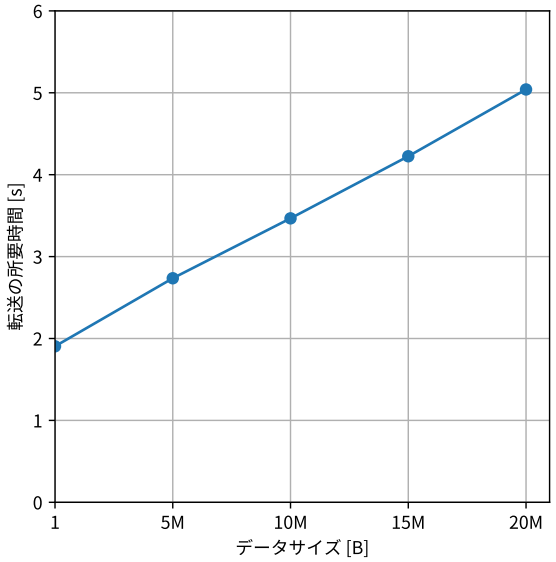

図 8 データ転送処理の所要時間

この結果、各コンテナ間で往復 20 ms の遅延が挿入される。

5.2.1 データの転送

SApp から RApp および RE へ自己破壊・出現データを転送する実験を行った。実験では、データの大きさを 1 B から 20 MB まで変化させた。SApp 側で転送を開始してから、RApp 側で暗号化されたデータをファイルに保存するまでの時間を測定した。各実験を 30 回繰り返し、その平均時間を求めた。

実験結果を**図 8** に示す。1 B の転送において、1.9 秒程度要している。1.9 秒のうち大部分（1.8 秒程度）は、remote attestation のための IAS への問い合わせに要する時間である。残りは、モノトニックカウンタの初期化処理と、データの暗号化および、ファイルシステムのアクセスにかかる時間である。

20 MB 程度のデータの転送に、5.0 秒要している。これは、30 Mbps 程度の転送速度であり、家庭用のインター

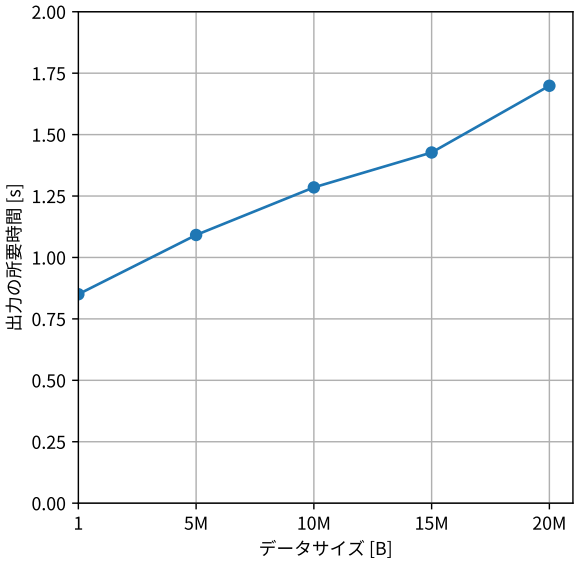

図 9 データ出力処理の所要時間

ネット回線でのダウンロード速度と同程度である。このことから、データの転送速度としては有用性があるといえる。

5.2.2 データの出力

RApp で自己破壊・出現データを出力する実験を行った。実験では、5.2.1 節で受信したデータを用いた。RApp で出力を要求してからそれが完了するまでの時間を測定した。各実験を 30 回繰り返し、その平均時間を求めた。出力条件として、真を返す次の式を評価した。

```
(< (now)
   (timevalue
     3000-01-01T00:00:00.0000Z))
```

データ出力処理の所要時間を**図 9** に示す。1B の出力において、0.85 秒程度かかっている。これは、モノトニックカウンタへのアクセス、データの復号および、ファイルシステムのアクセスにかかる時間である。20 MB のデータの出力を 1.7 秒で完了しており、有用性があるといえる。データの出力時間は転送時間よりも短い。その理由は、remote attestation を行っていないからである。

6. 関連研究

SafeVanish[11] およびその原型である Vanish[1] は、大規模な分散ハッシュテーブル (DHT) を利用した自己破壊データの実装である。これらのシステムでは、データの復号鍵を (k, n) しきい値法により n 個のシェアと呼ばれる素片に分け、別々に DHT に保存する。DHT を構成するそれぞれの計算機には鍵のシェアしか保存されず、復号のためには k 個のシェアの場所を知る必要がある。有効期限が過ぎたシェアは各ノードで破棄され、利用できなくなる。シェアを保管するノードのうち $(n - k + 1)$ 台が正しく破棄を行えばデータの復元が不可能になるため、全てのノードを信頼する必要がないのが優れた点である。

また、Emerge[2] は、同様に DHT を利用した自己出現データの実装である。こちらも erasure coding によってデータの復号鍵を秘密分散し、DHT の別々のノードに保存する。これらの断片を取り出すための経路は、指定された時刻まで知ることができない。

これらの手法では、DHT が巨大であることが前提として必要になるため、クラウドサービスの提供者には適した仕組みだが、個人の計算機における用途には適さない。また、データにアクセスする際には計算機が DHT に接続している必要がある。本研究では巨大な DHT を不要にする。また、オフラインでも利用可能にする。

また、タイムリリース暗号 [12] は、時刻を ID とした ID ベース暗号でデータが暗号化され、実際にその時刻が来たときに鍵生成局が対応する復号鍵を生成するという仕組みである。これは自己出現データを実現するためには良い仕組みだが、自己破壊データやアクセス回数に基づく出力条件は実現できない。また、本研究では、鍵生成局は不要であり、単なる時刻サービスがあれば十分である。

Fidelius[8] は、ブラウザや OS に対する攻撃からユーザの入力データを守るために、Web ページのセンシティブな部分のみを SGX enclave 内で実行する。また、信頼されたディスプレイおよび入力デバイスと、これらと enclave とのセキュアチャネルを実装している。畑ら [13] は、オンラインゲームのチート防止のため、ゲームのコアロジックを SGX enclave で実行している。これらの研究は、利用者によってアプリケーションが改ざんされることを想定している。これは本研究でも共通している。これらの研究では、自己破壊・出現データの実現方法を示していない。

7. 結論

この論文では、Intel SGX による自己破壊・出現データの実装について述べた。自己破壊・出現データとは、ある条件が満たされたときに限りアクセスできる仕組みをもったデータのことである。本研究で実現する自己破壊・出現データは、オフラインで利用でき、個人の計算機で完結する。実装で利用する時刻源、および、モノトニックカウンタは、汎用サービスとしても提供可能であり、データの送信者以外が提供することもできる。

提案方式に基づき、送信側のアプリケーション、受信側のアプリケーション、および、受信側の機密実行環境を Rust 言語で実装した。また、信頼された時刻とモノトニックカウンタを提供するサーバを Python 言語で実装した。

提案方式をセキュリティ面で解析した。その結果、ファイルに対するロールバック攻撃、メッセージに対するリプレイ攻撃および、コードの改ざんを検出可能で、不正にデータを出力されることはないことを確認した。データの

転送と出力に要する時間の測定を行った結果、所要時間は有用性があることがわかった。

今後の課題は、実際のアプリケーションに組み込み、評価することである。

参考文献

[1] Geambasu, R., Kohno, T., Levy, A. A. and Levy, H. M.: Vanish: Increasing Data Privacy with Self-Destructing Data, *18th USENIX Security Symposium (USENIX Security 09)* (2009).

[2] Li, C. and Palanisamy, B.: Emerge: Self-Emerging Data Release Using Cloud Data Storage, *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pp. 26–33, DOI: 10.1109/CLOUD.2017.13 (2017).

[3] Sabt, M., Achemlal, M. and Bouabdallah, A.: Trusted Execution Environment: What It is, and What It is Not, *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1, pp. 57–64, DOI: 10.1109/Trustcom.2015.357 (2015).

[4] Intel Corporation: Intel Software Guard Extensions (Intel SGX) (2020). [Online; accessed 27. Oct. 2020].

[5] Adams, C., Cain, P., Pinkas, D. and Zuccherato, R.: Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP), RFC 3161, RFC Editor (2001). [Online; accessed 2. Nov. 2020].

[6] Matetic, S., Ahmed, M., Kostiainen, K., Dhar, A., Sommer, D., Gervais, A., Juels, A. and Capkun, S.: ROTE: Rollback Protection for Trusted Execution, *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1289–1306 (2017).

[7] Intel Corporation: Introduction to Intel SGX Sealing (2020). [Online; accessed 27. Oct. 2020].

[8] Eskandarian, S., Cogan, J., Birnbaum, S., Brandon, P. C. W., Franke, D., Fraser, F., Garcia, G., Gong, E., Nguyen, H. T., Sethi, T. K., Subbiah, V., Backes, M., Pellegrino, G. and Boneh, D.: Fidelius: Protecting User Secrets from Compromised Browsers, *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 264–280, DOI: 10.1109/SP.2019.00036 (2019).

[9] 小清水滋，新城　靖，板野肯三，榮樂英樹，松原克弥：中立的 VMM による動画像を対象とした著作権保護，情報処理学会研究会報告，システムソフトウェアとオペレーティング・システム研究会 (OS)，2012-OS-123(8) (2012).

[10] Jones, M., Bradley, J. and Sakimura, N.: JSON Web Token (JWT), RFC 7519, RFC Editor (2015). [Online; accessed 2. Nov. 2020].

[11] Zeng, L., Shi, Z., Xu, S. and Feng, D.: SafeVanish: An Improved Data Self-Destruction for Protecting Data Privacy, *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 521–528, DOI: 10.1109/CloudCom.2010.21 (2010).

[12] 光成滋生：クラウドを支えるこれからの暗号技術，秀和システム (2015).

[13] 畑　輝史，オブランピエールルイ，河野健二：TEE によるスケーラブルかつチート耐性のあるオンラインゲームアーキテクチャ，情報処理学会研究会報告，システムソフトウェアとオペレーティング・システム研究会 (OS)，2020-OS-150(8) (2020).