情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2023-OS-158 No.10
2023/2/21

VMイントロスペクションを用いた TLS通信の監視

胎中 峻[1,a)]　石黒 健太[1,b)]　廣津 登志夫[1,c)]

概要：侵入防止システム (IPS) は増加するサイバー攻撃からサーバを保護するために重要な役割を担っている．IPS では外部とサーバの通信を監視することで攻撃の検知を行っているが，セキュアな通信のために利用される Transport Layer Security (TLS) プロトコルによる暗号通信に対しては，活用可能な情報が著しく減少する．この問題を解決するために，プロキシサーバや仮想マシンイントロスペクション (VMI) を用いて TLS 通信を復号する手法が提案されている．しかし，これらの手法には，それぞれ攻撃面の拡大や処理の遅延といった問題がある．本研究では，共有ライブラリのセマンティック情報を活用した, VMI による TLS 通信監視手法を提案する．ここでは，TLS の実装として広く用いられている OpenSSL の内部構造に着目することで，既存の VMI で遅延の原因となっていたメモリ空間の探索をなくし，処理遅延の削減を試みる．本提案を一般的に利用されている仮想マシンモニタである KVM と LibVMI ライブラリを用いて実装し，仮想マシン内で動作する OpenSSL を利用するサーバに対して監視を行い，監視処理による通信の遅延時間が削減可能であることを示す.

1. 序論

近年ではサイバー攻撃が増加するとともに，攻撃手法が高度化・複雑化し，世界中で被害が増加している．そのため，サーバなどを守るためにインターネット通信を監視・分析を行う Intrusion Prevention System(IPS) や Intrusion Detection System(IDS) などのセキュリティツールは重要な役割を担っている．しかし，セキュアな通信を実現するために用いられる Transport Layer Security(TLS) が暗号化した通信に対して，IPS/IDS は中身の詳細な内容を見ることができないため，十分な監視を行うことができない．そのため，TLS 通信がマルウェアなどの攻撃通信に悪用されると攻撃の検知ができず，被害が拡大する可能性がある．

この問題を解決するために TLS 通信の監視手法が提案されているが，それぞれの手法には問題が存在している．その一つの手法であるプロキシサーバを使用した手法は，送られてきた通信を復号して監視を行う．しかし，プロキシサーバはサーバとクライアント間の通信経路上に配置されるため，外部の攻撃者がプロキシサーバに対して攻撃通信を送ることが可能であり，攻撃により監視活動を停止させられる可能性がある．別の手法としては，監視対象であるサーバを仮想マシン (VM) 上で動作させ，仮想マシンイントロスペクション (VMI) を用いて仮想化レイヤから通信を監視する手法が考えられる．しかし，この手法は仮想マシンのメモリ中から TLS 通信に利用される鍵情報を総当りで探索するため，VMI によって鍵を探索している間は VM を停止させる必要があり，TLS 通信自体に遅延が発生してしまうという問題がある．

本研究では，後者の仮想化レイヤから TLS 通信を安全に監視するシステムについて処理時間を改善する．ここでは，TLS 通信の実装に広く用いられている OpenSSL の内部構造の情報を利用し，通信を行っている VM のメモリ中に存在している通信内容を直接仮想化レイヤから取得することで，先行研究で問題になっていた通信の遅延の削減を試みる．本提案を一般に利用されている仮想マシンモニタである KVM と LibVMI ライブラリを用いて実装し，VM 内で動作する OpenSSL を利用するサーバに対して通信内容の監視を行い，発生した通信の遅延を測定することで評価を行う．

以下，第 2 節では IPS/IDS，プロキシサーバを用いた TLS 通信の監視手法の問題点を述べる．第 3 節では，VMI 技術の利点と関連研究について説明する．第 4 節では，本研究が想定している脅威モデルを述べる．第 5 節と第 6 節では本研究で提案する設計と実装について述べ，第 7 節では実装したシステムの実験と評価を行う．第 8 節で考察を

1 法政大学
Hosei Uniersity
a) shun.tainaka.3p@stu.hosei.ac.jp
b) kenta.ishiguro.66@hosei.ac.jp
c) hirotsu@hosei.ac.jp

ⓒ 2023 Information Processing Society of Japan

述べ，最後の第 9 節で本研究の結論を述べる．

2. TLS 暗号通信の監視手法

IPS/IDS における TLS 通信監視の問題を説明し，さらにプロキシサーバによる TLS 通信への対応とその問題点について述べる．

2.1 IPS / IDS による TLS 通信監視

侵入検知システム (Intrusion Detection System, IDS) は，システムやインターネット通信の監視を行い，外部から攻撃通信が確認された場合に，システム管理者に通知する監視システムである．IDS が不正な通信の検知，通知までを行うのに対し，侵入防止システム (Intrusion Prevention System, IPS) では，攻撃通信の防御までを行う．IPS/IDS の監視システムは，通信パケットの中身を監視し，分析することで攻撃通信かどうかを判断する．しかし，TLS を用いて暗号化された通信に対しては，暗号化されていない情報である送信元と宛先の IP アドレスやポート番号などのパケットヘッダから取得できる情報以外には監視の足がかりになる情報は得られない．これらの情報だけでは攻撃通信かどうかの判断において，監視精度の低下は否めない．

2.2 プロキシサーバによる TLS 通信監視

プロキシサーバを用いて TLS 通信を監視する手法は，IPS/IDS の監視手法の問題である TLS 通信を監視できない問題を解決しており，現在では広く利用されている．プロキシサーバは，自身が配置されるネットワーク上で動作しているサーバの代理としてインターネット上のクライアントからのアクセスを受け付ける役割を持っており，通信フィルタリングやログの取得などの IPS/IDS 機能を用いてサーバの保護を行っている．TLS 通信の監視を行うために，プロキシサーバは TLS 通信を行うクライアントとサーバの通信経路上に中継地点として配置され，それぞれの間に TLS 通信を行うためのセッションを張る．そして，送信元から送られてくる暗号通信をプロキシサーバが受信し，一度平文に復号して通信内容を確認する．そして，平文を再度暗号化してプロキシサーバが送信先へ送信することで，通信間の TLS による機密性を保ちながら通信の監視を実現している．

しかし，このプロキシサーバを用いた方法にはリスクがある．前述の通り，プロキシサーバは通信を行うクライアントとサーバの間に配置され，クライアントからの通信を受け付けている．そのため，外部の攻撃者はプロキシサーバに対して攻撃通信を送ることができてしまう．もしプロキシサーバに脆弱性が存在し，攻撃の影響を受けてしまった場合，プロキシサーバを通過する通信を改竄や盗聴などの被害が発生してしまう．

3. 仮想マシンイントロスペクション (VMI) による TLS 通信監視

以上の問題に対して本研究では，仮想化レイヤから保護対象を監視するシステムに注目する．この構成を採用した研究 [1] では，保護対象である TLS サーバを VM 上で動作させて，その VM の外部である仮想化レイヤから仮想マシンイントロスペクション (VMI) を用いて TLS 通信を復号する手法を提案した．

3.1 仮想マシンイントロスペクション (VMI)

仮想マシンイントロスペクションとは，動作している VM 内部の情報を VM の動作を管理する役割を持つ仮想マシンモニタ (VMM) や別の VM から覗き見る技術である．この技術は，VM が利用している CPU やメモリ，レジスタなどのリソース状況を取得することにより，VM 内部の監視や分析を可能にする．VMI を利用して VM の動作を監視する例として，VM のシステムコールの実行監視 [2] や関数呼び出しの監視 [3] が挙げられる．また，システムコールや関数呼び出しに渡されるパラメータを分析することで，VM に感染したマルウェアを検知する研究 [4] も行われている．VMI を実行するツールとして，LibVMI [5] や Volatility [6] などがある．

VMI が近年注目されている理由として，VM と VM の外部 (VMM など) の環境が乖離していることが挙げられる．近年ではサイバー攻撃の高度化に伴い，攻撃対象の上で動作している IPS/IDS などの監視ツールの無効化や改竄を行う攻撃が増えている．そこで，監視対象を VM 上で動作させて，VMM などの VM の外部で監視ツールを動作させることで，攻撃から監視ツールを保護するシステムが提案され [7]，注目された．このシステム構造では，VM と VM の外部 (VMM など) の環境が仮想化技術によって，ネットワークやハードウェアが論理的に隔離される．この環境の乖離性により，監視対象である VM が攻撃されたとしても，VM の外側で動作する監視ツールを攻撃から保護することが可能であり，正常な監視を継続することができる．

3.2 VMI を用いた監視処理による通信遅延の問題

VMI を用いて TLS 通信監視システムを提案している先行研究 [1] では，VMI を用いて TLS 通信の暗号化に用いるマスターキーと呼ばれる鍵を VM のメモリ中から探し出し，TLS 通信を復号する手法を提案している．サーバとクライアント間でやり取りされる TLS 通信の準備であるハンドシェイク通信を観測することで，VM のメモリ中にマスターキーが存在するタイミングを計り，メモリのスナップショットを VMI を用いて取得している．その後，スナップショット内のバイト列から正しく TLS 通信を復号できるかどうかを総当りで検証することにより，マスターキー

となる正しいバイト列を探し出している．

この手法では，TLS 通信の監視に必要な VMI 処理によって発生してしまう通信の遅延が問題として挙げられる．遅延が発生する原因は，VM のメモリのスナップショットを取得している間 VM を停止させるからである．先行研究ではこの処理遅延を抑えるために，取得するスナップショットのサイズを縮小する工夫を行っている．これは，マスターキーが生成される前後のタイミングを TLS 通信の準備として送られるハンドシェイク通信の SH (Server Hello) メッセージと CSP (Change Cipher Spec) メッセージを観測することで計り，その間に新しく書き込まれた，もしくは書き換えられた VM のメモリ領域のみをスナップショットとして取得している．ここでは実際に Apache2 プロセスが単一の TLS 通信を行う際の監視処理遅延を測定し，取得するスナップショットのサイズ約 104MB に対して約 8.7 ミリ秒の通信遅延が発生するという結果を得ている．この遅延は，スナップショットを取得するメモリ空間が小さい最善に近い場合の結果であり，TLS 通信を行うプロセスが複数実行されている場合や，該当するプロセスが利用するメモリ空間のサイズが大きい場合など，取得するスナップショットのサイズが大きくなってしまう際にはより大きな遅延が発生する．また，監視対象システムに侵入したマルウェアが大きな通信の遅延から監視されていることに気づき，自身の活動を抑制するなどして，監視を回避する可能性もあると述べられている．

3.3 VMI を用いた OpenSSH 通信監視

先行研究 [8] では，VMI を用いた SSH 通信の監視手法を提案した．この手法は，SSH 通信の実装に広く利用されている OpenSSH の内部構造の情報を使用して，VM のメモリ中に存在するマスターキーの場所を特定してピンポイントに VMI を行っている．実装では，OpenSSH が利用する ssh オブジェクトや session_state オブジェクトに注目している．これらのオブジェクトは，マスターキーをはじめとする SSH 通信に必要な情報が格納されている．そして，これらのオブジェクトに格納されているメンバのオフセットからマスターキーが格納されている VM 中のメモリの位置を特定して，ピンポイントに VMI を行うことで直接マスターキーを取得する手法を実装した．さらに，VMI を行うタイミングと利用するオブジェクト情報を取得するために，マスターキーを生成する OpenSSH 関数の呼び出しを VMM から検知する手法を実装した．これは，監視する関数に VMI を用いて INT3 命令を挿入して，INT3 命令が実行された際に発生するブレークポイント (BP) 例外を VMM が検知することで実現している．さらにこのタイミングで，レジスタに引数として渡されるオブジェクト情報を取得している．この研究では，クライアントからサーバへの接続時間を監視している場合と監視していない場合のそれぞれで測定を行い，結果を比較した．サーバに負荷がかかっていない最良状況で実行した結果として，監視を行っている場合は，監視を行っていない場合の約 1.4 倍の接続時間となり，約 50 ミリ秒の通信遅延が発生している．

4. 脅威モデル

脅威モデルとして，TLS 通信を行うサーバが動作しているゲスト OS とホスト OS は信頼できる．そして攻撃者は，TLS 通信を行うクライアントを自由に操作することができ，サーバとの通信を行いサーバの脆弱性を攻撃することで権限の奪取などを行うことを想定している．

5. 設計

本研究では，第 2 章で述べた IPS/IDS，プロキシサーバでの TLS 通信監視の問題を解決する仮想化レイヤから監視するシステムの構築を目的としている．さらに，第 3 章で述べた研究 [1] の問題点であった TLS 通信の監視処理による通信の遅延の削減削減を目指す．ここでは関連研究 [8] の手法を参考にし，TLS 通信の実装に広く用いられている OpenSSL の内部構造の情報を利用して，TLS 通信に利用されるマスターキーもしくは通信内容を直接 VM のメモリ中から取得する手法を提案する．

OpenSSL [9] は，SSL/TLS に関する様々な機能を提供しているオープンソースのライブラリであり，多くのサービスの通信の暗号化に使われている．そのため，OpenSSL の関数レベルで暗号化通信の解析を可能にすることができれば，多数のサービスやアプリケーションに対して適用することが期待できる．

OpenSSL には TLS 通信で使用するマスターキーの生成や証明書の作成，暗号化と復号などの API 関数を提供しており，それらの関数には引数として，マスターキーや復号した結果を格納するメモリ領域へのアドレスが渡される．そのため，引数の情報を取得することができれば，その引数が示すメモリ領域を取得することにより，マスターキーや通信内容を直接取得できる．一般的に，関数が呼ばれる際にはレジスタかスタック領域に引数が積まれるため，関数が呼ばれた直後に VM が利用するそれらのリソースから引数を取得することが可能である．

以上の情報を利用し，今回提案するシステムを設計した．図 1 はそのシステム設計を示している．TLS 通信を監視する手順は次のとおりである．

(1) VMM から VM に対して VMI システムを実行し，通信内容が監視できるような準備を予め施す．
(2) VM のサーバが外部クライアントと TLS 通信を開始する．
(3) VM のサーバのプロセスで VMI を実行するタイミングの目印となる関数が呼ばれたことを VMI が検知する．
(4) VMM が VMI システムを実行し，VM のメモリ内から鍵情報，もしくは通信内容を取得する．

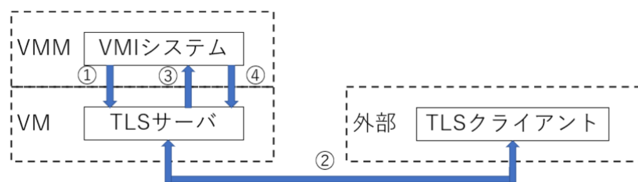

図 1　システム設計図

6. 実装

今回，SSL_read 関数に注目して通信内容を直接取得する仕組みを実装した．本章では，SSL_read 関数から通信内容を取得する方法の説明と SSL_read 関数が呼ばれたタイミングを VMM で検知する手法，そして実際の通信内容取得の際に行った手順について述べる．

実装環境として，ホスト OS とゲスト OS は共に ubuntu20.04 を用い，仮想化エミュレータは一般的に利用されている KVM を用いた．KVM は，Linux に組み込まれた仮想化テクノロジーであり，KVM を使用すると，ホスト OS である Linux を VMM として動作させることができる．VMI プログラムには，LibVMI というライブラリを用いて，C 言語で実装を行った．

6.1 LibVMI ライブラリ

LibVMI [5] は VMI を実現する C 言語ライブラリである．LibVMI が対応する仮想化プラットフォームは Xen と KVM，Qemu であり，VMI を行う対象のゲスト OS は，Linux と Windows に対応している．提供する機能は，VM のメモリの読み取りと書き込み，VM の CPU レジスタへの読み取り，書き込み，VM の一時停止と再開などがある．LibVMI は，ゲスト OS に関する情報 (カーネルが参照するシンボル情報やプロセス情報を保持している構造体の構成情報）を予め取得しておき，VMM に配置しておくことが必要になる．この情報を使用することで，LibVMI はセマンティックギャップを解消し，VMI を実現している．Linux の場合であると，カーネルが参照するシンボル情報が格納されている System.map ファイルとプロセスを管理する task struct 構造体のデータ構造の情報が該当する．

6.2 SSL_read 関数からの通信内容取得

本研究では，実装として OpenSSL の SSL_read 関数に注目した．この関数は，特定の SSL/TLS 通信のコネクションに対して送られてきた通信内容を指定したバッファ領域に読み取る関数であり，OpenSSL を用いた TLS 通信の実装で通信を受信する際に呼び出される．引数として，通信コネクションを管理する SSL オブジェクト，通信内容を読み取る先へのバッファ領域のポインタ，通信内容として読み取るバイト数の 3 つが渡される．本研究では，この SSL_read 関数の第 2 引数であるバッファ領域のポインタを利用して，TLS 通信の内容を取得する．

第 4 節で述べた通り，関数が呼ばれる際にはそのプロセスが使用する仮想メモリ中のスタック領域かレジスタに引数が格納される．SSL_read 関数の場合であると，第 1 引数の SSL オブジェクトは RDI レジスタ，第 2 引数のバッファ領域へのポインタは RSI レジスタ，第 3 引数のバイト数は RDX レジスタにそれぞれ格納される．そのため，SSL_read 関数が呼ばれた直後に RSI レジスタからバッファ領域へのポインタを示す第 2 引数を VMI を用いて取得し，さらにその引数が示すポインタに対して VMI を行うことで通信内容が格納されているメモリ領域を覗きみるという手順を踏む．

6.3 ブレークポイントによる SSL_read 関数のフック

SSL_read 関数が呼ばれた直後に VMI を実行しなければ，レジスタの内容が更新されてしまい引数情報を正確に取得できないため，通信内容を取得することができない．そのため，SSL_read 関数が呼ばれた直後のタイミングを VMM 側で検知しなければならない．本研究では，共有ライブラリの特性を利用し，SSL_read 関数の先頭にブレークポイント (BP) を配置して，その BP を VMM 上の VMI プログラムが検知することによって VMM 側で SSL_read 関数の実行されるタイミングを取得する．OpenSSL は共有ライブラリであり，複数のプロセスが物理メモリ上の一箇所にのみ存在するコードを共有してそれぞれの実行を行う．そのため，一度共有ライブラリに対して中身の変更を行うと，そのライブラリは物理メモリ上に展開されている限り呼び出され続け，呼び出し元の各プロセス上で変更した通りに実行される．この特性を利用し，SSL_read 関数の実行検知を行う．ブレークポイントは，実行フローの特定のポイントでプログラムの実行を中断できるようにするものであり，デバッグをする目的で用いられる．ブレークポイント例外を発生させるのが，INT3 命令であり”0xcc”の 1 バイトのオペコードである．この INT3 命令を SSL_read 関数の先頭のコードである endbr64 命令の部分に上書きを行い，SSL_read 関数が呼ばれた直後にブレークポイント例外が発生するようにする．endbr64 命令とは，プログラム実行に呼ばれた関数が安全かどうか判断するために用いられる命令であり，実際の関数の動作には関係しないため，endbr64 命令を INT3 命令へ上書きする対象とした．実装では，endbr64 命令である”f3 0f 1e fa”の 4 バイトを，何も実行しない命令である NOP 命令”0x90”と INT3 命令を組み合わせた”cc 90 90 90”の 4 バイトに書き換えた．

6.4 通信内容の取得手順

図 2 は通信内容を取得する手順を示している．実際の手順は以下のようである．

(1) 準備として，SSL_read 関数の先頭コードである endbr64

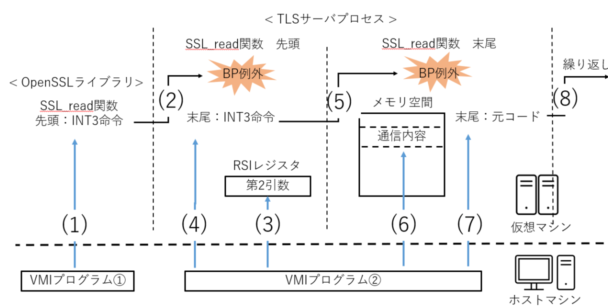

図 2　通信内容取得の実行手順

命令”f3 0f 1e fa”の部分に INT3 命令と NOP 命令”cc 90 90 90”を挿入する．

(2) ゲスト OS が TLS 通信を開始し，SSL_read 関数に挿入した INT3 命令が実行され，BP 例外が発生する．
(3) ホスト OS の VMI システムが BP 例外を検知し，SSL_read 関数の第二引数を RSI レジスタから取得する．
(4) さらに，SSL_read 関数の末尾の部分に 2 個目の INT3 命令を挿入する．
(5) SSL_read 関数が実行が再開され，末尾の部分の INT3 命令が実行され，再度 BP 例外が発生する．
(6) 取得した第 2 引数が示すメモリ領域を取得することで，通信内容を取得する．
(7) 末尾の INT3 命令を元のコードに戻し，実行が再開される．
(8) 監視サイクルが繰り返される．(2) へ

VMI の実行は 2 度行う必要があり，1 度目は SSL_read 関数が呼ばれた直後に RSI レジスタから引数を取得するタイミングである．2 度目は通信内容が新しくメモリに格納された後の SSL_read 関数の実行が終わる直前で，通信内容を取得するタイミングである．1 度目は，事前に SSL_read 関数の先頭に INT3 命令を挿入することで，タイミングを計ることは述べた．2 度目の VMI を実行する SSL_read 関数の終了直前のタイミングを計るために，最初に BP 例外が発生したタイミングでさらに SSL_read 関数の末尾のコードの部分に対して 2 個目である INT3 命令の挿入を行う．そして，通信内容をメモリから取得した後，末尾のコードを挿入した INT3 命令から元のコードに戻しておくことで，正常に SSL_read 関数が実行を終え，この通信内容を取得する監視サイクルが繰り返し行えるようにしている．

7. 実験と評価

はじめに，実験環境での VMI プログラムの実行速度を評価するために，LibVMI を用いて VM の物理メモリ全体の読み取り時間を測定した．測定環境は，CPU は Intel (R) Core (TM) i7 - 7700 CPU @ 3.60 GHz × 8，メモリは 16 GB，OS は 64bit ubuntu 20.04.4 LTS，カーネルは 5.4.24 +である．1024MB の VM に対して測定した結果，約 6 秒

```
(gdb) disass SSL_read
Dump of assembler code for function SSL_read:
   0x00007ffff7c84b20 <+0>:     int3
   0x00007ffff7c84b21 <+1>:     nop
   0x00007ffff7c84b22 <+2>:     nop
   0x00007ffff7c84b23 <+3>:     nop
   0x00007ffff7c84b24 <+4>:     sub    $0x18,%rsp
   0x00007ffff7c84b28 <+8>:     mov    %fs:0x28,%rax
   0x00007ffff7c84b31 <+17>:    mov    %rax,0x8(%rsp)
```

図 3　IN3 命令を挿入後に呼び出された TLS サーバプログラムの SSL_read 関数の先頭オペコード

```
(gdb) x/32bx 0x7fffffffd6a0
0x7fffffffd6a0: 0x68    0x65    0x6c    0x6c    0x6f    0x0a    0x8d    0x4d
0x7fffffffd6a8: 0x93    0x78    0x12    0xe4    0x8c    0x73    0x15    0x3c
0x7fffffffd6b0: 0xb0    0xf7    0x89    0x04    0x0e    0xc1    0xe2    0xef
0x7fffffffd6b8: 0x2f    0x86    0x90    0xff    0x1b    0xd6    0xfd    0xa8
```

図 4　ホスト OS で通信内容を取得した様子

```
LibVMI init succeeded!
Monitoring start
Waiting for BPevents...
client -> server : 68656c6c6f
```

図 5　ゲスト OS で確認した結果

かかった (10 回測定した平均時間)．

次に，TLS 通信の内容取得を行った．VMI システムを動作させるホスト OS の上に，TLS 通信を行う 2 つのゲスト OS を用いて実験を行う．監視対象となるゲスト OS には TLS サーバプログラム，通信相手となるゲスト OS には TLS クライアントプログラムを動作させて通信を行い，ホスト OS で VMI システムを起動させて通信内容を取得した．事前に，監視対象のゲスト OS に対して SSL_read 関数の先頭に INT3 命令を仕込み，クライアントとサーバの間で通信セッションを繋いだ後，クライアントからサーバへ向けて”hello”という内容の通信を送り，通信内容の取得した．通信内容が正しく取得できているかを確認するため，ゲスト OS 内で GDB を用いて通信内容を格納しているメモリをダンプして比較した．

監視の準備として SSL_read 関数の先頭に INT3 命令が挿入した結果を，GDB を用いて確認した．監視対象のゲスト OS で，サーバプログラムの SSL_read 関数に対して逆アセンブルを行い，先頭の endbr64 命令”f3 0f 1e fa”が INT3 命令と NOP 命令”cc 90 90 90”に上書きされていることが確認できた (図 3)．VMI プログラムを実行して通信内容を取得した結果，ASCII コードで”hello”を意味する”68 65 6c 6c 6f”のバイト列を取得した (図 4)．さらに，GDB で通信内容を格納する領域に対してメモリダンプを行った (図 5)．その結果，ホスト OS で取得したバイト列と同じであることが確認でき，実装した VMI システムが通信内容を正確に取得できていることが確認できた．

さらに，通信監視を行っている場合と行っていない場合の SSL_read 関数の実行時間を比較することにより，発生する通信の遅延を測定した．クライアントから送信される通信内容は，5 文字の文字列である．監視を行っていない

ⓒ 2023 Information Processing Society of Japan

場合，SSL_read 関数の実行時間は約 0.03 ミリ秒であった (100 回測定した平均時間)．一方，監視を行っている場合，SSL_read 関数の実行時間は約 1.536 ミリ秒であった (100 回測定した平均時間)．以上から，通信監視による通信の遅延は約 1.533 ミリ秒発生し，SSL_read 関数の実行時間は VMI により監視している状況では通常の時と比べ，約 50 倍かかることがわかった．

8. 考察

先行研究 [1] では本研究と同じように，LibVMI を用いて仮想マシンの物理メモリ全体を読み取る時間を測定することで，実行環境での VMI プログラムの実行速度を評価している．結果は，メモリサイズ 1024MB の VM に対して 2.4 秒である．本研究での VMI 実行時間が約 6 秒であったため，先行研究と比べて約 2.5 倍ほど遅い実行環境であることがわかる．

この実行環境で通信内容を取得した際の通信遅延は約 1.5 ミリ秒であり，先行研究の約 8.7 ミリ秒の通信遅延と比べて約 7.2 ミリ秒，約 83%通信の遅延を削減した．測定した環境での VMI 実行時間が先行研究の約 2.5 倍遅いことから，より遅延を削減することが期待でき，素直に 2.5 倍の差を考慮すると約 94%の遅延削減となる．さらに，先行研究の約 8.7 ミリ秒の通信遅延は，取得するスナップショットのサイズが小さいような，測定状況が最善の場合での結果であるため，測定状況次第ではさらに遅延が増加すると考えられる．一方，提案した手法では測定状況に左右されずに，常に約 1.5 ミリ秒での遅延に抑えることができる．

先行研究での手法は，理論上は TLS 通信の実装手段にとらわれずに通信の監視を行うことができるのに対し，本提案手法は OpenSSL で実装した TLS 通信のみに対して監視が可能であり，理論上は汎用性において先行研究の方が優れていることになるが，現実的には多くのサービスやアプリケーションが OpenSSL を利用しており，本提案手法の適用範囲も十分に広いと考えられる．さらに，提案した SSL_read 関数に注目した手法であると，通信を受信するたびに VMI を行い通信内容を取得するため，遅延の発生頻度が高い．一方，先行研究では一度マスターキーを取得すると，その後の通信は外部で取得する通信を復号することにより監視できるため，VMI を行う必要がない．そのため，マスターキーが作成，または更新されるタイミングのみ VMI を行えば良いため，通信遅延の発生頻度が少ない．しかし，本提案手法は個々の SSL_read 関数が遅延している分，遅延時間は一定で小さいため，監視による遅延なのかサーバの性能限界なのかの区別が付きにくい．一方で，先行研究では一時的に大きな遅延が発生することから，監視状況にあることをマルウェアが容易に推測できる．

加えて，今回は通信内容を格納する領域を VMI で取得することで通信の監視を行ったが，本提案手法と同じように OpenSSL の内部情報を利用してマスターキーを取得することで通信の監視を行うことができる．この手法であれば，通信遅延の発生頻度は先行研究と変わらないため，より通信遅延が少ない監視を実現できると考えられる．マスターキーを取得する方法として，OpenSSL が通信を行う際に利用する SSL オブジェクトや SSL_SESSION オブジェクトの情報の利用が考えられる．SSL オブジェクトは，セッション情報を管理する SSL_SESSION オブジェクトへのポインタを保持しており，SSL_SESSION はマスターキーを保持している．そのため，通信に必要な関数の引数として渡される SSL オブジェクトから SSL_SESSION オブジェクト，マスターキーとポインタを辿っていくことで，マスターキーを取得することができると考えている．この手法での VMI の実行内容は，本提案手法と大きく変わらないため，発生する通信遅延も本研究での測定結果と同程度の約 1.5 ミリ秒であると推測している．

9. 結論

本研究では、IDS/IPS やプロキシサーバ，VMI による TLS 通信の監視手法での問題に対応するため，TLS 通信の実装に用いられる OpenSSL の内部構造の情報を利用して，通信内容を直接 VM のメモリから取得するシステムを提案した．LibVMI ライブラリを用いて、SSL_read 関数の引数から通信内容を格納する VM 内のメモリ領域を特定し、通信内容を取得するシステムを実装した．実験の結果，通信内容の取得は成功し，先行研究と通信の遅延を比べることで，遅延を削減できる可能性を示した．

謝辞 本研究は，JST CREST JPMJCR21M4 の支援を受けたものである．

参考文献

[1] Taubmann, B., Frädrich, C., Dusold, D. and Reiser, H. P.: TLSkex: Harnessing virtual machine introspection for decrypting TLS communication, *Digital Investigation*, Vol. 16, pp. S114–S123 (online), DOI: https://doi.org/10.1016/j.diin.2016.01.014 (2016). DFRWS 2016 Europe.

[2] Sentanoe, S., Taubmann, B. and Reiser, H. P.: Virtual Machine Introspection Based SSH Honeypot, *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, SHCIS '17, New York, NY, USA, Association for Computing Machinery, p. 13–18 (online), DOI: 10.1145/3099012.3099016 (2017).

[3] Sentanoe, S., Taubmann, B. and Reiser, H. P.: Sarracenia: Enhancing the Performance and Stealthiness of SSH Honeypots Using Virtual Machine Introspection, *Secure IT Systems* (Gruschka, N., ed.), Cham, Springer International Publishing, pp. 255–271 (2018).

[4] Mishra, P., Aggarwal, P., Vidyarthi, A., Singh, P., Khan, B., Alhelou, H. H. and Siano, P.: VMShield: Memory Introspection-Based Malware Detection to Secure Cloud-Based Services Against Stealthy Attacks, *IEEE Transactions on Industrial Informatics*, Vol. 17, No. 10, pp. 6754–6764 (online), DOI: 10.1109/TII.2020.3048791 (2021).

ⓒ 2023 Information Processing Society of Japan

[5] Payne, B. D.: Simplifying virtual machine introspection using LibVMI., (online), DOI: 10.2172/1055635.
[6] Foundation, V.: Volatility Framework – volatile memory extraction utility framework (2007).
[7] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Destributed Systems Security Symp.*, pp. 191–206 (2003).
[8] Sentanoe, S. and Reiser, H. P.: SSHkex: Leveraging virtual machine introspection for extracting SSH keys and decrypting SSH network traffic, *Forensic Science International: Digital Investigation*, Vol. 40, p. 301337 (online), DOI: https://doi.org/10.1016/j.fsidi.2022.301337 (2022). Selected Papers of the Ninth Annual DFRWS Europe Conference.
[9] The OpenSSL Project: OpenSSL: The Open Source toolkit for SSL/TLS (2003). www.openssl.org.

ⓒ 2023 Information Processing Society of Japan