Computer Security Symposium 2018
22 - 25 October 2018

システムを改変しない攻撃者とOS機能を信頼の基点とする暗号処理の安全性に関する一考察

渡辺 大[1]　吉田 直樹[2]　坂本 純一[2]　宮地 遼[2]　松本 勉[2]

概要：モバイル端末を使った金融決済サービスが注目を集めるなど、IT システム上で取り扱う情報の価値が高くなっている。これに伴い、攻撃のリスクが高まっている。この脅威に対抗するために、ハードウェアのサポートを得ながら暗号処理をセキュアな領域で実行する TrustZone 等の技術が開発されているが、我々はソフトウェアだけで同様の機能を実現する手段について検討を行っている。本稿では、「システムに改変を加えない攻撃者」という比較的能力が低い攻撃者を定義する。また、Android OS や PC Linux などの UNIX 系 OS を例にとり、その妥当性について検討を行った結果について報告する。

キーワード：モバイルセキュリティ, ソフトウェア保護

A note on system respecting adversary and the security of cryptographic process under well-behaved operation system

Dai Watanabe[1]　Naoki Yoshida[2]　Junichi Sakamoto[2]　Ryo Miyachi[2]　Tsutomu Matsumoto[2]

Abstract: The trend of cumputation platform for various services is moving from such decicated hardware to more publicly available hadware such as mobile-phone and public/private cloud system, and the Internet. This change indicates that the security risks increases; vulnerability of software, malware infection, theft of terminal devices, and so on. Some hardware supported countermeasures against these threats have been proposed so far such as ARM TrustZone or Intel SGX. Contrary, we started the research on purely software based countermeasures. In this report, we define a *system respecting adversary* which does not modify the system softwares. In addition, we show some circumstantial evidences by investigating the architecture of UNIX-like OSs such as Android OS and PC Linux, and malwares running on it.

Keywords: Mobile security, software protection

1. はじめに

1.1 背景

計算基盤として、パブリッククラウドとモバイル端末の普及が進んでいる。そして、これらをインフラと捉え、従来の専用ハードウェアに基づくサービスを、クラウド基盤の上に構築することで、低コストかつ利便性の高い IT サービスで置き換える動きが活発になっている。

金融決済を例とすると、従来であれば、金融機関の支店などに設置されている専用端末（ATM）にキャッシュカードを差し込み、PIN コードなどでユーザ認証を行ってから、振り込み操作などを行っている。これに対し、モバイル決済サービスでは、スマートフォンなどの汎用デバイス上にインストールされた決済用アプリケーション（以下、決済アプリ）を用いてクラウド上の決済サービスを利用する。現時点では、ユーザ認証は、モバイル端末のログイン認証等で代替されることが多いが、将来的には、内蔵 NFC を用いた IC カード認証や生体認証を組み合わせた多要素認証に置き換えられていくものと考えられる。

情報セキュリティの観点からユーザ端末を見ると、マル

[1] (株) 日立製作所, 〒 244-0817 神奈川県横浜市戸塚区吉田町 292
[2] 横浜国立大学, 〒 240-8501 神奈川県横浜市保土ケ谷区常盤台 79-1

ウェアの高度化、標的型攻撃の深化により、完全な防御は難しいという見解が一般的になっており、セキュリティ対策の立案においても、攻撃者に侵入されることを前提とした対策に重点が置かれるように変化してきている。セキュリティベンダの McAfee のレポート [1] によれば、モバイル端末のマルウェア感染率は 2014 年度で約 10%を維持しており、攻撃と対策が拮抗したまま、完全な駆除には成功していないことを示している。

このように、ユーザ端末のセキュリティ対策が十分とは言えない一方で、上記のようなサービスの多様化に伴い、価値の高い情報（たとえば金融決済システムのユーザ認証情報）が IT システム上で取り扱われるようになっている。これは、IT システムを攻撃して秘密情報を入手することが、従来以上に魅力的になっていることを意味しており、これまで以上に高度な攻撃が展開されることが危惧される。

1.2 先行研究

このような課題に対して、暗号処理など高いセキュリティが要求される処理を仮想的に別領域で行う ARM TrustZone [3] や処理中の RAM を保護する Intel SGX [2] が開発されている。

また、特定のハードウェアに依存せずに、ソフトウェア実装技術だけで同様の機能を実現するための研究も存在する。white-box cryptography (WBC) は Chow らが提案した攻撃者モデル (および対策実装技術) である [8]。WBC の攻撃者は、暗号処理を実行している環境の内部状態 (CPU のレジスタや RAM 上のデータ) を任意のタイミングで観測／改ざんすることができる。Chow らの研究を皮切りに、多くの対策実装が提案されているが、WBC の攻撃者は非常に強力であり、残念ながら安全と言える実装は存在しないのが現状である [11]。

Bogdanov らは、端末に侵入するマルウェアと、マルウェアと通信しながら鍵復元を行う攻撃者モデル (gray-box cryptography: GBC) を提案した [9]。GBC モデルでは、マルウェア自身の能力は制限されており、また、マルウェアを含む攻撃者が利用可能なストレージの量に上限が設定されている。したがって、大量のストレージを利用することで、GBC モデルにおける安全性を実現することができる [9], [10]。

この他にも、大石らはプログラムを構成する各々の命令の実行タイミングやコード中での配置場所に着目し、デバッガを利用可能な攻撃者であっても、時間や場所が一定以上離れている場合には、これらの情報を繋げることができない、とした [7]。大石らは、この仮定の元で、プログラムの改ざんを検出する方式を提案している。

また、渡辺は、Intel SGX と同様の機能をソフトウェアだけで実現可能かという観点での検討を行っている [12]。Intel SGX では CPU as a black-box (攻撃者は CPU 上のデータにアクセスできない) というモデルを仮定し、CPU (レジスタ) 上のデータを RAM に置く場合に暗号化する仕組みを提供しているが、渡辺はハードウェア的なサポートが無い場合には、OS の割り込み処理で CPU 上のデータが強制的に RAM に退避されることに指摘し、その対策を提案している。

1.3 本稿の貢献

[12] における検討では、攻撃者の能力を抽象的に捉え、プログラムとデータに対して能動的／受動的にアクセスできるかという観点で分類している。たとえば、上記の RAM 暗号化はプログラム、データに対して受動的にアクセスする攻撃者を想定した対策であり、プログラムを改変するような攻撃者に対する対策ではない。その一方で、[12] で提案された RAM 暗号化方式の安全性は OS のソフトウェア割り込みが正しく実行されることを前提としており、攻撃者が (対象プログラムではなく) システムを改変する場合には、主張している安全性の根拠が失われる。

そこで、本稿ではまず「システムに改変を加えない攻撃者 (System Respecting Adversary)」という、比較的能力が低い攻撃者を定義する。また、Android OS や PC Linux などの UNIX 系 OS のアクセス制御機構や実際のマルウェアの能力について調査し、上記の攻撃者が妥当なものであることを確かめると共に、対象プログラムの使用する RAM 領域へのアクセス方法を明らかにする。

1.4 本稿の構成

本稿の以降の構成は以下の通りである。まず、第 2 節で想定環境や UNIX 系 OS のプログラム実行の基本的な事項を説明し、本稿が想定する攻撃者を定義する。次に、想定する攻撃者の妥当性を検証するために、第 3 節と第 4 節で UNIX 系 OS 一般および Android OS のメモリアクセスについて、また、第 5 節ではマルウェアの攻撃手法を紹介する。この後で、第 6 節で想定する攻撃者の妥当性を議論する。最後に第 7 節で本稿をまとめる。

2. 準備

2.1 前提とする機器構成と攻撃者の目的

本研究では、主にスマートフォンなどのモバイル端末を対象としている。モバイル端末は汎用的な計算機として CPU、RAM/ROM などのメモリ、ハードディスクや SSD などのストレージ装置に加えて、カメラ等のセンサデバイス、無線通信装置を供えている。なお、本研究は将来的に TrustZone を含む HSM (Hardware Security Module) の利用も検討範囲に入れるべきであるが、現時点では環境をある程度限定して議論を進めるため、HSM の存在は仮定しないものとする。

従来のブラックボックスモデルでは、装置間でやりとり

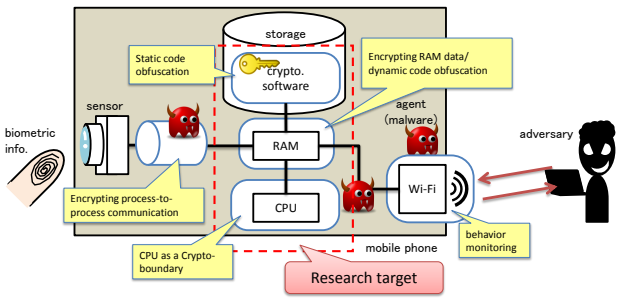

図 1 Malware infected threat model and possible countermeasures

されるデータは攻撃者に漏れないことを想定している。しかし、マルウェアに感染している端末では、攻撃者は装置間通信のためにセンサのバッファや RAM 上に置かれたデータを取得可能である。すなわち、端末内部のデータバスや RAM は公開通信路に近い状態と考えられる。

本研究で対象とする攻撃者の目的は、攻撃対象機器に含まれた何らかの秘密情報を取得することに限定する。DoS 攻撃や、DDoS 攻撃のためのボット化は対象外とする。図 1 に示すように、攻撃対象となるシステムにはバイオメトリック情報や復号鍵等の秘密情報が格納されており、あるプロセスで利用時に RAM 上に展開される。攻撃者の目的は別のプロセスからこの秘密情報を取得することである。このような RAM 上のデータにアクセスしてくる攻撃者に対して、Intel SGX (Software Guard Extension) のような対策が実現されている。Intel SGX では、CPU からの入出力を攻撃表面と捉え、CPU とは独立にプロセスを管理し、それぞれの実行環境を隔離する。また、攻撃表面の外側にある RAM にデータを書き込む場合は、暗号化でデータを保護する。

2.2 想定する攻撃者

Intel SGX が想定する安全性モデルは、ハードウェアによるサポートを前提としていることから、適用範囲が限定的である。これに対し、渡辺は別プロセスの RAM を参照する際には必ずソフトウェア割り込みが発生することに着目し、ソフトウェアだけで SGX と類似の暗号境界を実現する手法を提案している [12]。この手法は、OS が制御する割り込みのタイミングが重要であることから OS を改変するような攻撃者に対しては安全ではない可能性がある。そこで、本稿では [12] の提案手法の安全性を評価する第一歩として、比較的弱い攻撃者を新たに定義する。

定義 1 システムを構成するプログラムを、システムコールなどカーネルモードで実行される原始的な機能 (システムプログラム) と、ユーザモードで実行されるその他の機能 (ユーザプログラム) に分ける。システムプログラムを書き換えない攻撃者を**システムを改変しない攻撃者** *(system respecting adversary)* と呼ぶ。

また、本稿ではこれに加えて、攻撃者がデバッグ機能を利用できないものと仮定する。この仮定が現実的であることは、6 節で述べる。

3. UNIX 系 OS のメモリアクセス

本節では、UNIX 系 OS においては RAM アクセスの際に必ずソフトウェア割り込みを介すことと、その際に利用される 3 つのシステムコールを解説する。

3.1 UNIX 系 OS におけるプログラム実行とアクセス制御

UNIX 系の OS では、一般のプロセスはユーザーモードで実行される。しかし、ファイルへの読み書きなどカーネルが管理するデバイスへのアクセスではカーネルモードの権限が要求される。この問題を解決するための仕組みがシステムコールである。システムコールは、ユーザーモードで実行されるプロセスがデバイスにアクセスするための API である (図 2 参照)。ここでは、UNIX 系 OS のシステムコールの仕組みについて概説する。詳細については、たとえば [17] を参照されたい。

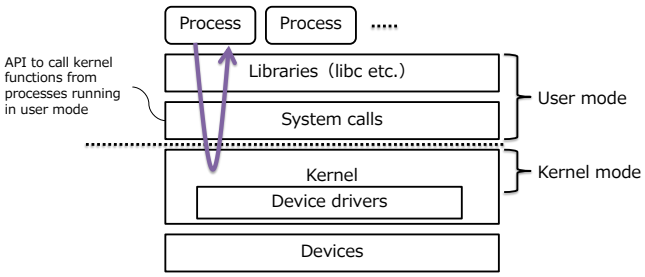

図 2 System call and mode change

カーネルはシステムコールを受け取ると所定の処理を行う。近年の CPU では、モードの切り替えを行うための命令を持っており、Intel x86 アーキテクチャでは `int` 命令が、ARMv8 では `svc` 命令がこれに相当する。以下、ARMv8 の場合について、例外処理の実行について説明する。まず、`svc` 命令が実行されると、CPU はモードをユーザーモードからカーネルモードに切り替え、特定のアドレスに飛ぶ。次に、割り込みベクタテーブルを読んで、システムコール番号が指示する処理にジャンプする。レジスタ上のシステムコール番号と引数をカーネルスタックに `push` する (それ以外のレジスタも適宜退避する)。レジスタの退避が済んだら、システムコールサービスルーチン (システムコールの処理本体) を実行する。システムコールサービスルーチンが終了したら、カーネルスタックから退避したデータを復帰、CPU をユーザーモードに戻して元のアドレスに戻る。

Intel や ARM が提供する CPU の多くは、アクセス制御機能を提供している。ARMv8 では、4 段階のレベル (例

外レベル EL0〜EL3) が設定されており、OS のカーネルモードは EL1、一般的なアプリは EL0 で実行される。ハイパーバイザを実行する EL2、セキュアモニタを実行する EL3 は本稿の範囲外となる (図 3 を参照)。また、ARM と Linux の組み合わせであれば、ユーザーモードとカーネルモードの境界が EL0 と EL1 の境界に相当するので、以降では CPU の提供するアクセス制御機能には触れず、OS のモード切り替えの観点のみで考えれば良い。

3.2 メモリアクセスとシステムコール

UNIX 系 OS では数百のシステムコールが用意されているが、RAM データの取得および書き込みに関するシステムコールは以下の 3 つである。

`ptrace` 親プロセスによる別のプロセスの実行の監視/制御、コアイメージ (core image) やレジスタの調査/変更、ブレークポイントによるデバッグ、システムコールのトレースを行う。

`read` ファイルディスクリプタが参照するファイルからバッファへ読み込む。

`write` バッファからファイルディスクリプタが参照するファイルへデータを書き込む。

3.2.1 プロセスのメモリ空間と関連するファイル

UNIX 系 OS では、プロセスが利用しているメモリもファイルとして管理されている。メモリに関連する主要なファイルを表 1 にまとめる。

`/proc/PID/`以下に納められている一群のファイルは、プロセス番号が `PID` のプロセスに関するシステムファイルである。メモリに関するファイルは、メモリマップ情報を管理する `maps`、メモリそのものを表すファイル `mem`、仮想物理アドレスと物理アドレスの変換テーブル `pagemap` の 3 つである。このうち、`/proc/PID/mem` は `ptrace` を経由してアクセスすることが前提となっており、(`read`/`write` を介した) 通常の方法では読み書きできない。

`/dev/mem` はシステムが管理するメモリ空間そのものであり、物理アドレスでメモリを管理している。これに対して、`/dev/kmem` はカーネルが利用するメモリ空間であり、仮想アドレスでメモリを管理している。`/dev/mem` はルート権限があれば読み書き可能である。

`/dev/kmem` にはカーネルスタックが含まれていると考えられ、これにアクセスできれば、割り込み発生時に退避された CPU 上のレジスタ等の値を読み書きすることができる。

4. Android OS のアクセス制御

この節では、本稿で対象とする Android OS におけるメモリのアクセス制御を解説し、通常のプロセスは他のプロセスのメモリ空間を参照できないことを示す。しかしながら多くの Android マルウェアに対してはこの前提は適用できず、秘密情報を秘匿するためにはメモリを保護する仕組みが必要になることを次節で示す。

4.1 Android OS のアーキテクチャ

Android OS はスマートフォンやタブレットなどの携帯情報端末上での動作を想定して Google 社により開発された Linux をベースとする OS である。既存の Linux カーネルにミドルウェア、ユーザインターフェース、ソフトウェアアプリケーションを加えたパッケージとして提供している。Android OS のアーキテクチャを図 4(右) に示す。

Android OS の構成をより単純にモデル化すると、Linux カーネル、カーネルが管理するプロセスとして稼働する仮想マシン、および仮想マシン内で実行されるユーザアプリといった 3 種類のフレームワークと考えることができる。単純化したモデルを図 4(左) に示す。

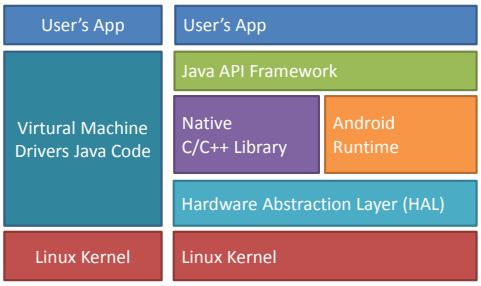

図 4 Android OS Architecture (Left: Simplifed, Right: Reference)

下層で動作するカーネルは既存の Linux をベースとして、小リソースのデバイスでも動作するように改変を施したものが用いられている。そのため上層のソフトウェアは基本的な Linux の機能を用いることができ、ハードウェアドライバやシステムコールなどの OS のコア技術を使って端末を制御することが可能である。しかし Android OS が動作する携帯情報端末の多くは、通常のデスクトップ Linux に比べ著しく計算能力が低いため、それに応じて改変を行っている部分も存在する。例えばデスクトップ Linux で一般的に用いられている C 言語標準ライブラリの GNU C Library (glibc) は、Android OS では実装されておらず、代わりにサイズの縮小を図った Bionic C Library と呼ばれる C ライブラリが実装されている。これにより通常の Linux 環境で使用できるシステムコールが利用できない場合があり、システムコール以外にも、セマフォや共有メモリなどを使う関数が実装されていない場合がある。glibc と bionic C の詳細な違いについては文献 [16] に譲る。

上層では、Android OS のユーザアプリを動作させる仮想マシンと仮想マシン内部で動作するユーザアプリにより構成される。この仮想マシンは DalvikVM (version 5.0) /Android RunTime (version 5.x 以降) と呼ばれ、Java バ

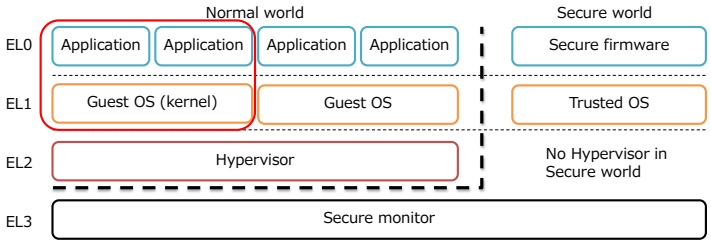

図 3 Exception levels in ARMv8 Architecture [4]

表 1 System files related to memory (PID shoudl be replaced by the process number of the target process)

File	Access control	Outline
/dev/mem	740	Whole memory space
/dev/kmem	–	Virtual memory space used by kernel
/proc/PID/maps	444	Memory map of the process identified by PID
/proc/PID/mem	600	Memory space used by the process identified by PID
/proc/PID/pagemap	400	The translation table between the physical and the virtual memory address for the process identified by PID

イトコードにより動作する。Android OS のユーザアプリは一般的にこの仮想マシン上で動作しアプリ開発の多くは Java を用いて開発を行う (2017 年より Java と完全互換可能な “ Kotlin ” と呼ばれる言語が使用可能になっている)。

4.2 Android OS のメモリ管理

各ユーザアプリを動作させる仮想マシンは、カーネルが管理する仮想メモリ上に領域が確保されスワップやページング、割り込み処理といった一般的なメモリ管理をカーネルが担当する。カーネルが管理するメモリと仮想マシンの関係を図 5 に示す。カーネルが行う低級のメモリ管理は通常の Linux と同様のメモリアクセスを提供する。この時、カーネルが処理しているのは 1 次元のシーケンシャルな仮想メモリであり、カーネルは 1 つの仮想マシンを 1 つのプロセスとみなして仮想マシンにメモリを割り当てる。

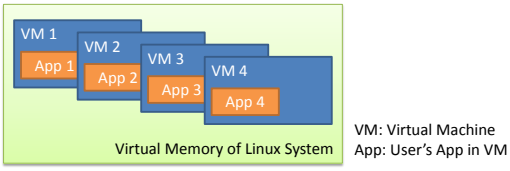

図 5 Memory area is devided for each virtural machine and managed

4.3 Android OS のメモリアクセス制御

Android OS は OS としてコアとなるメモリアクセス機能は Linux カーネルのメモリ管理に依存しながら、ユーザアプリを動かす仮想マシン内部にもアクセス管理機構を持っている。Android OS の仮想マシンが行うアプリのメモリアクセスを図 6 に示す。カーネルが取り扱う仮想メモリ上に、起動しているユーザアプリを動作させる仮想マシンが常駐し、各仮想マシンが内部でユーザアプリを処理する。

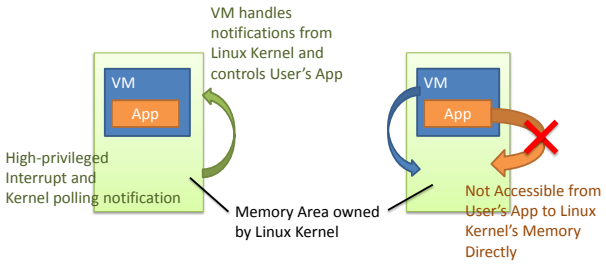

図 6 Android OS controls memory access of User's App

各ユーザアプリはカーネルが処理しているメモリ空間を見ることができず、自分に割り当てられている仮想マシン外にアクセスするにはデバイス管理者の許可が必要になる。また許可が付与された場合でも、ユーザアプリから仮想マシン外へのアクセスは仮想マシンの API を利用する形でしか利用できないため、実質ユーザアプリが直接自分を動かすか仮想マシンを越えてメモリの読み書きを行うことはできない仕組みになっている。

5. マルウェアの攻撃手法

5.1 一般的なマルウェアの攻撃手法

近年のマルウェアはその多くが標的プラットフォームの

ネットワークリソースを使うものに分類され、トロージャンとなりユーザの個人情報を不正に収集したり、端末をボット化し DoS (Denial of Service) 攻撃などの踏み台として利用されたりすることが多い [14], [15]。本研究で調査した範囲では、Android OS を標的とするマルウェアが別のユーザアプリに成りすまし、自身を動かす仮想マシンを越えて他のユーザアプリの監視や改竄を行うことは少なく、通常の権限内で攻撃を完了する場合が多い。

一方で、他のユーザアプリや仮想マシンを改ざんするような挙動を示すマルウェアの大半は、Linux カーネルのルート権限を取得することを攻撃の起点とする。カーネルより上層で動作するすべてのソフトウェアは、すべてカーネル管理下のメモリ上で動作するため、ルート権限を取得したマルウェアは、通常の Linux 管理者がプロセスを監視・制御するのと同様のプロセスで Android OS のユーザアプリを管理できる。近年では、他のユーザアプリを標的とするマルウェアの多くが、ルート権限を取得することに重きを置いているものが大半となっている。

5.2 Android 上の特権を持つ攻撃者

昨今ではスマートフォンを標的とする悪性アプリやユーザによるリバースエンジニアリングなどによって引き起こされる「ルート化」がセキュリティ上の問題となっており、ルート化された端末は Linux カーネルのシステムドメインの管理者権限を持つことが可能となる。例えば、ユーザアプリに成りすましたマルウェアが Android OS のルート化に成功すると、本来仮想マシンを越えてアクセスできない筈のメモリ領域にアクセスが成功となり、カーネルレベルで端末を制御できるようになるため、別のユーザアプリの盗聴や改竄を行うことが可能となる (図 7)。

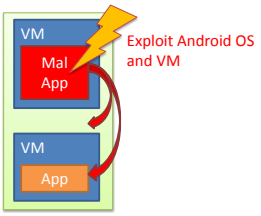

図 7 Malware exploits Android OS to access over VM's access control

またカーネル側から来る割り込み処理やポーリングの通知については仮想マシン側で処理を行い、ユーザアプリ自身が明示的に割り込み時の処理を定義しなくても、メモリの退避やユーザアプリの復帰に必要な処理を行うよう Android OS 側で実装されている。

5.3 ptrace をはじめとする Linux カーネルのシステムコール

前節で述べたように、Android 上の特権を持つ攻撃者は別プロセスのメモリ領域にアクセス可能だが、この際のメモリアクセスにも 3.2 節で示した 3 つのシステムコールのどれかを介す必要がある。従って、割り込み直前のタイミングの RAM 暗号化によって、SGX と類似の暗号境界をソフトウェアだけで実現できる可能性がある。RAM だけでなく CPU レジスタの値も取得できる ptrace システムコールの存在には注意しなければならないが、4.1 節で示したとおり Android OS では通常の Llinux で一般的に実装されているカーネルの機能が一部使用できないことがある。中でも ptrace は一般的な Linux カーネルに実装されているシステムコールでありながら、Android OS の多くで使用できないシステムコールの一つである。ptrace は Linux システム上で生成されたプロセスを別プロセスから監視・制御を行う際に用いられるシステムコールであり、一般的にはブレークポイントによるデバッグやシステムコールのトレースに使用される。しかし、Android OS では一般的にユーザがカーネル領域からユーザアプリを管理することはほとんどないため、各デバイスのベンダが Android OS のビルド・インストール時に ptrace をはじめとするいくつかのシステムコールを削除している場合がある。しかし、システムコール自体はカーネル側で実装されるものであり、標準ライブラリ (binonic C Library) 内部で関数が用意されていなくても、カーネル側では残っている場合もある。その場合 C/C++などネイティブコードを生成する開発環境において別途参照可能なシステムコールのライブラリを記述し、再び Android OS をビルド・インストールできれば Android OS 内部でも該当のシステムコールを復元できる可能性がある。

6. 考察

6.1 デバッグ機能を利用しない攻撃者の現実性

5.3 節で述べたように、Android ではデフォルトで ptrace システムコールを利用できない。

PC Linux では通常 ptrace システムコールを利用可能である。また、カーネルが利用するメモリ空間/dev/kmem については、Linux 2.6.26 以降では、カーネル設定オプション CONFIG_DEVKMEM が有効になっている場合のみ/dev/kmem を利用できる。一例として著者らが確認したところ、Raspberry Pi3 用の OS である Raspbian Linux では/dev/kmem は存在しなかった。このように、PC Linux においても、ptrace システムコールおよび/dev/kmem の利用を停止することが可能であり、これら 2 つの対策を施すことで、攻撃者の能力を/dev/mem 経由で RAM にアクセスするだけに制限することができる。

このように、Android OS や PC Linux では、「デバッグ

機能を利用できない」という仮定が比較的容易に構築できるので、システムを改変しない攻撃者はソフトウェア割り込み時の退避データにアクセスできない。つまり、攻撃者は CPU のレジスタ上にあるデータにアクセスできないので、Intel SGX で提唱された CPU as a Crypto-Boundary と同様のセキュリティ境界を (弱い攻撃者に限定はするものの) ソフトウェアだけで実現できる。

6.2 想定する攻撃者がメモリにアクセスする方法

図 8 は、Android 上において、特権を持つ攻撃プログラムが攻撃対象のプログラムの使用する計算リソースにアクセスする手段を表している。システムを改変しない攻撃者を仮定すれば、特権を持つ攻撃プログラムであってもユーザーモードで動作している。

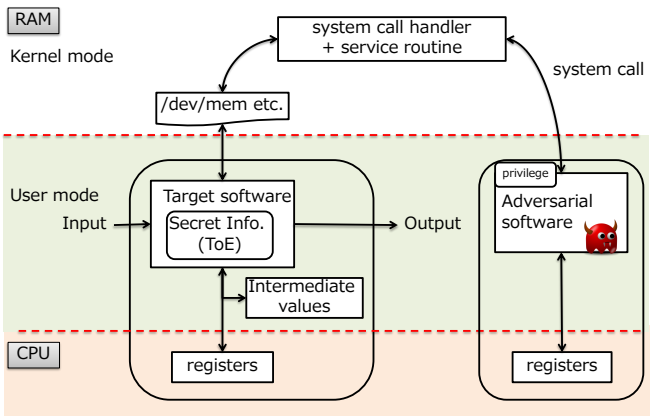

図 8 Adversarial model on Android OS

攻撃プログラムが他のプログラムが利用する計算リソースにアクセスするためには、`read/write` や `ptrace` などのシステムコールを発行し、システムファイルにアクセスすることになる。`ptrace` を利用できないことを仮定すれば、`read/write` を介して`/dev/mem` にアクセスするのが、攻撃に利用可能な唯一の手段になると考えられる。

7. まとめ

7.1 結論

本報告では、計算機環境として Android 端末を、攻撃者として、特権 (ルート権限) を取得した上でシステムの提供する機能を利用して目的を遂行する攻撃者を想定した。この想定は、現在のマルウェアの能力からすれば妥当であると考えられる。

また、このような条件下では、「攻撃者は割り込み発生時に CPU 上のデータを取得できる」という想定は現実的ではなく、攻撃者は`/dev/mem` 経由で RAM にアクセスするだけで、CPU レジスタ上に配置した秘密情報を入手することができないと考えられることがわかった。

7.2 今後の課題

本報告では、攻撃者の能力を定性的な観点から考察した。しかし、攻撃者がどの程度の頻度で、どの程度のメモリ領域にアクセスできるのか、という観点について詳しいことはまだわかっていない。今後、防御策の有効性や効率を論じる上で、攻撃者の能力を定量的に評価することは重要であると考える。

参考文献

[1] マカフィー株式会社, “McAfee 脅威レポート：2014 年第 4 四半期 2014 年 10 月から 12 月のセキュリティ脅威の調査結果を報告,” 2014. `http://www.mcafee.com/jp/threat-center/report/download89.aspx` (2016/09/15 閲覧)

[2] H. Matthew, “Intel SGX for Dummies (Intel SGX Design Objectives),” `https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx`, 30 November, 2015 (2017 年 9 月 11 日閲覧).

[3] ARM Ltd, “TrustZone Technology Overview.” Available at `http://www.arm.com/products/esd/trustzone_home.html`.

[4] ARM Cortex-A Series Programmer’s Guide for ARMv8-A Version: 1.0, 24 March, 2015. Chapter 3. `http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A\_v8\_architecture\_PG.pdf`

[5] National Institute of Standards and Technology (NIST), “Advanced Encryption Standard (AES),” Federal Information Processing Standards, FIPS 197, November 2001. `https://doi.org/10.6028/NIST.FIPS.197`.

[6] Peter Schwabe and Ko Stoffelen, “All the AES You Need on Cortex-M3 and M4,” Cryptology ePrint Archive: Report 2016/714, 2016. `https://eprint.iacr.org/2016/714`.

[7] 大石和臣, 松本勉, “ソフトウェアの自己破壊的タンパー応答により達成し得る機能改変困難性,” 2011 年 暗号と情報セキュリティシンポジウム (SCIS2011), 4E2-2, 2011.

[8] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot, “White-Box Cryptography and an AES Implementation,” In Kaisa Nyberg and Howard M. Heys, editors, Selected Areas in Cryptography, volume 2595 of Lecture Notes in Computer Science, pages 250–270, Springer, 2002.

[9] Andrey Bogdanov and Takanori Isobe, “White-Box Cryptography Revisited: Space-Hard Ciphers,” ACM Conference on Computer and Communications Security 2015, pp. 1058–1069, 2015.

[10] Pierre-Alain Fouque, Pierre Karpman, Paul Kirchner, and Brice Minaud, “Efficient and Provable White-Box Primitives,” Cryptology ePrint Archive: Report 2016/642, 2016.

[11] CHES 2017 Capture the Flag Challenge, `https://whibox.cr.yp.to/` (2017 年 10 月 30 日閲覧).

[12] 渡辺 大, 暗号機能のソフトウェア実装におけるマルウェアの脅威に関する一考察, 信学技報, vol. 117, no. 316, ICSS2017-44, pp. 35–40, 電子情報通信学会, 2017.

[13] “Android Developers プラットフォームアーキテクチャ,” `https://developer.android.com/guide/platform/index.html`. Last Visited 2018/03/05.

[14] Malwarebytes 2017 state of malware report, `https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf`. Last Visited 2018/03/01.

[15] Exproit Database Offensive Security's Exploit Database Archive, https://www.exploit-db.com/. Last Visited 2018/03/05.
[16] Matheu Devos, BIONIC vs GLIBC REPORT, Faculty Industrial Sciences &Technology Master Thesis. Available on http://irati.eu/wp-content/uploads/2012/07/bionic_report.pdf.
[17] Daniel P. Bovet and Macro Coesati, 高橋 浩和 監訳, 杉田 由美子, 高杉 昌督, 畑崎 惠介, 平松 雅巳, 安井 隆宏 訳, 詳解 Linux カーネル 第 2 版, オライリー・ジャパン, 2003.