平成 26 年度 修士論文

OS の管理者権限機構の改良による ユーザ環境移行手法の検討と実装

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1353034　吉原 大夢

主任指導教員　多田 好克　教授
指導教員　小宮 常康　准教授
指導教員　新谷 隆彦　准教授

提出日　平成 27 年 1 月 26 日

目次

図目次

表目次

ソースコード目次

第 1 章

はじめに

近年、スマートフォンやタブレット等の携帯端末が普及してきている。これらの携帯端末は、従来のフィーチャーフォンと呼ばれる携帯電話端末に比べてパーソナルコンピュータに近づいており、無線 LAN による高速なネットワーク通信やタッチパネルを活かした表現力豊かなユーザインタフェース等が提供されている。これらの機能は端末利用者それぞれの使い方に合わせて設定できるため、従来に比べてスマートフォン等の携帯端末では、無線 LAN 設定やシステム設定などの設定項目が多様化している。

また、スマートフォンのシェアの大部分を占める Google の Android や Apple の iOS 等のモバイル OS では、電話やメールを始めメモやアラーム、ゲーム等がアプリとして提供されている。端末に初めから含まれる端末開発会社のアプリだけでなく、サードパーティ製のアプリの開発も活発になっている。Android では GooglePlay、iOS では AppStore といった公式マーケットを持っており、開発者はそれぞれのマーケットに開発者登録をすることで自らの開発したアプリを世界中に配信することができる。端末利用者はこれらの中から、自分の必要な機能を持った好みのアプリを端末にインストールして利用することができる。そのため、利用者の端末毎にインストールされるアプリの種類や数は千差万別である。

フィーチャーフォンに比べて、一般開発者がアプリ市場に参入しやすいため、アプリの多様化が進んでおり、利用者によってカスタマイズされた、その利用者に取って最適な『ユーザ環境』とも呼ぶべきものの多様化も進んだ。ユーザ環境は、端末利用者の利用している環境に合わせて構築される。例えば、利用している無線 LAN や利用しているアプリ、ホーム画面におけるアプリの配置、アラームの設定等、ユーザ環境は日々利用していく中でユーザそれぞれが使いやすい状態に設定変更されている。

しかしながら、近年のモバイルの OS やハードウェアの進歩は目覚ましく、アプリの動作条件等を満たすために数年以内に端末を買い替えることが通常となっている。例えば、Google 製の Android リファレンス端末である Nexus では、OS アップデートのサポートを発売から 18 ヶ月までとしている [4]。端末を買い替えた場合には当然、新しい端末の上に自分の使い慣れたユーザ環境を同じように再構築する、もしくは使い慣れた環境を捨てて新たに一からユーザ環境を構築していくことになる。ユーザ環境というのは、利用者が

端末を利用してきた結果であるため、利用期間に応じて大きくなると考えられる。そのため、前者の場合には、前の端末の利用に慣れているほど再構築の手間がかかってしまう。さらに、長期に渡って利用した端末であれば、細部の設定等を忘れてしまっている可能性がある。また、高齢者等のデジタル機器に疎い人は、他人に設定を依頼することもよくあるため、端末所有者本人が設定を把握していない可能性もある。そのため、端末利用者自身が今まで利用していた環境を再現することが困難な場合がある。一方で後者の場合には、使い慣れない環境に慣れなければならず、これもまたユーザにとって望ましいことではないだろう。

一般的にこのような手間を削減するためのツールとして、端末の設定などをバックアップする機能やアプリが提供されている。しかし、一般的な端末で利用できるアプリからではアクセス権が与えられていないがためにアクセス出来ないデータが多数存在する。そのため、バックアップを作成できる範囲が限られており、本研究で言うところのユーザ環境の移行が出来ているとは言い難い。

これに対して、管理者権限を取得できるようにする Android の root 化や iOS の JailBreak と呼ばれる手法がある。これを適応した端末を対象としたバックアップアプリも存在する。このようなアプリは、管理者権限を利用して端末内の隅々のデータにアクセスができるため、上で述べたアクセス権の問題は起きない。しかし root 化や JailBreak は、端末のセキュリティを甘くする、メーカのサポート対象外となってしまう等のデメリットがあるため、一般の端末利用者に対して推奨できるものではない。

他にも、携帯端末とパーソナルコンピュータを接続して環境移行を提供するアプリも存在する。このようなアプリを利用するためには、パーソナルコンピュータ側にも環境移行のための設定を行わなければならない。そのため、バックアップのための事前準備のステップが多く使い勝手の良いものではない。

そこで本論文では、オープンソースとして提供されているモバイル OS である Android を対象としたユーザ環境を移行する手法について検討する。これまでの考察を踏まえて、Android の管理者権限機構を改良し、セキュリティレベルを維持したまま管理者権限を取得できるようにする。同時に、それを利用してユーザ環境の移行を実現するアプリの実装を行う。また、パーソナルコンピュータ等は使わず移行前後の端末のみを使うようにすることで、ユーザがスマートフォン端末を買い替えたその場ですぐにユーザ環境の移行が行えるような手軽なツールの実装を目指す。

本論文では 2 章にて提案手法や関連研究、関連ツール等の本研究の背景について説明した後、3 章にて本研究で扱う Android の構造や概念について説明する。その後、4 章にて提案するツールの設計について議論し、5 章で実装について説明する。6 章にて実装したツールの評価・考察を行い、最後にまとめと今後の展望について述べる。

第 2 章

研究背景

2.1 研究目的

本研究では、端末利用者の手間を出来る限り削減することができるユーザ環境移行ツールの作成を目的とする。図 2.1 に提案ツールのイメージを示す。

提案するツールは、Android のプリインストールアプリとして実装し、ユーザは端末を購入したらすぐに環境移行を実行できるような、簡易に扱える物とする。しかし、詳しくは 2.3 節で述べるが、一般的なアプリではアクセス権がないという問題があるため端末内の全てのデータを取得することができない。

そこで本研究ではまず、管理者権限機構の改良によって、3.2 節で述べる root 化よりも比較的安全に管理者権限を扱えるようにする。そして、これを利用することによってユーザ環境の移行を実現する。

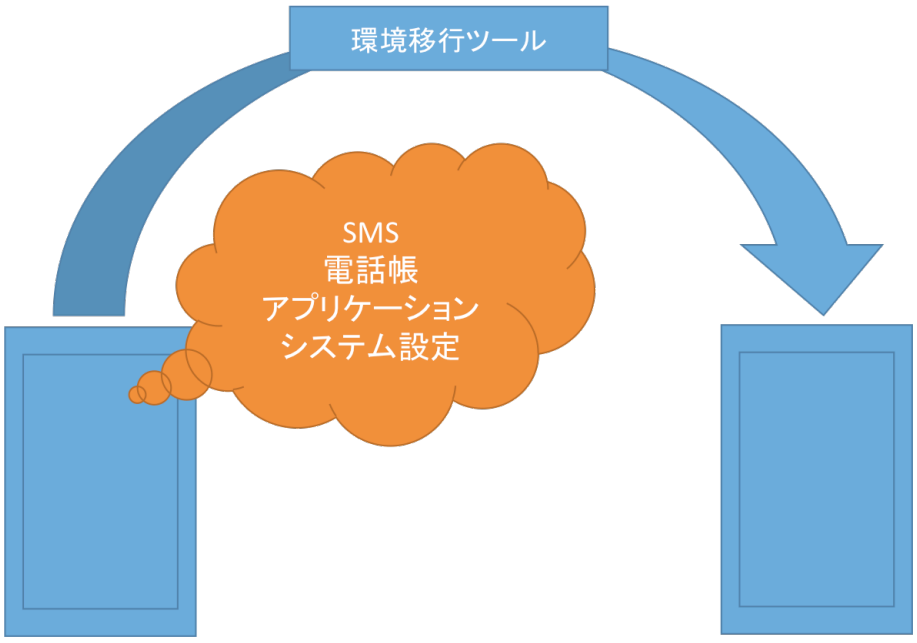

図 2.1 提案手法イメージ

2.2 関連研究

本節では、Android のセキュリティに関する先行研究を紹介する。本研究では、端末中のほとんどのデータを扱うため Android のデータ構造やセキリティを考慮しなければならない。Android のセキュリティについては様々な視点から議論されており、システムの仕組みを理解するためや安全なアプリを作るためには非常に有用である。提案するシステムでは特にパーミッションを利用してセキュリティを保証するため、ここでは特にパーミッションに関連する研究について紹介する。

2.2.1 パーミッション承認の問題点

Android アプリからカメラや電話帳の読み書き等の端末の特定の機能を利用するためには、事前に利用する機能をパーミッションとして宣言しなければならない。そして、端末利用者がアプリをインストールする際に、そのアプリがどの機能を使うのかを明示し端末利用者に承認を求める。このシステムによって、そのアプリの機能からして不要だと思われるパーミッションを持つ場合には、悪意のあるプログラムであると疑うことができる。その一方で、現行のシステムではアプリのインストール時に開発者の求めるパーミッションをすべて承認しなければならず、一部のパーミッションのみを許可してのインストールやインストール後にパーミッションの許可を変更することができない。Mohammad Nauman らはこの欠点に着目し、パーミッションを一部のみ許可・拒否することができるシステムの改良案 [1] を提案した。

2.2.2 システムに検知されないパーミッションの問題点

James Sellwood らは、Android に用意されているパーミッションについて調査 [2] を行った。Android の開発サイクルは非常に早く、仕様が大幅に改定されることも頻繁にある。その中で、Android の機能の増加とともにパーミッションも増えていくが、同時に廃止されるパーミッション等システムに認知されないパーミッションも存在する。認知されていないパーミッションはシステムから無視されてアプリがインストールされる。Android アプリのインストールの仕組み上、一度インストールされたアプリへのパーミッションの承認の変更は認められない。そのため、このようなパーミッションが含まれるアプリをインストールし、Android の仕様が変更されてこのパーミッションがアクティブになった場合には、端末利用者が意図しない動作をすることがあるという脆弱性があることを示した。

2.2.3 アプリインストール時におけるパーミッションの問題点

David Barrera らは、Android アプリを大量に集めてデータセットを作り、それを利用した Android アプリのインストールにおけるセキュリティメカニズムについて実環境に近い状態での解析した [3]。その結果として、現状の Android のセキュリティメカニズムの 1 つであるパーミッションシステムにセキュリティや柔軟性の観点から問題点があることを指摘し、現行のシステムから大きな変更を行うことなく実現可能な改善案を示した。

2.3 関連ツール

端末のデータを収集して保存するという意味においては、バックアップツールは本研究で提案するシステムに類似する。そこで本節では、既存のバックアップツールを概観する。

既存のバックアップツールにはいくつか種類があり、アプリとして実装しているものや OS の機能として組み込まれているもの等がある。以降でそれらについて紹介し、その後本研究で提案する手法との比較を行う。

2.3.1 一般的なバックアップアプリ

Google Play[5] には、現状様々なバックアップアプリが公開されているが、これらは端末内の一部のデータを対象としている。これは、アプリから端末内のデータへのアクセスが制限されており、アプリから取得できないデータが存在するためである。詳しくは 3.1 節で述べる。

本研究の目指すところは、Android 搭載端末利用者のユーザ環境の移行である。そのため、一般的なバックアップツールでは移行可能なデータが少なく、ユーザ環境の移行と呼ぶには至っていないと言える。

2.3.2 root 化端末対象のバックアップアプリ

同じく Google Play[5] に公開されているアプリの中にも root 化済対象のバックアップアプリというものが存在する。root 化とは、詳しくは 3.2 節で述べるが、AndroidOS 搭載端末の管理者権限 (root 権限) を取得可能な状態にすることを指す。2.3.1 で述べた一般的なバックアップアプリの場合には制限されていたデータへのアクセスが、管理者権限を利用することによって可能になる。そのため、全データの取得が可能になり、端末内のデータのフルバックアップが可能となる。root 化端末対象のバックアップアプリとしては Titanium Track が配布している Titanium Backup[6] が有名である。

しかし、端末を root 化することは、セキュリティを甘くすることやメーカのサポート対象外になること等のデメリットもあり、一般的な端末利用者が気軽に利用できるものではない。

2.3.3 開発者用ツールのバックアップ機能

Android のバックアップを行うことができるツールの 1 つに、Android Debug Bridge(以下 ADB と呼ぶ) が挙げられる。ADB は、Google が Android 開発者に配布しているデバッグツールである。ADB は開発に利用しているパーソナルコンピュータと Android 端末を繋ぎ、パーソナルコンピュータから Android 端末のデバッグログの表示やアプリのインストール・アンインストール、Android 端末の shell への接続など様々な機能を指示する。

Android4.0 への更新時にその中の機能の 1 つとして、Android のバックアップ/リストアを行う機能が追加された。

```
$ adb backup
```

上記の様なコマンドをパーソナルコンピュータから実行することによって、Android 端末を root 化せずにフルバックアップを作成することが可能である。

一方で、ADB はあくまで開発者ツールとして配布されており、操作はコマンドラインを利用する。一般的な端末利用者にとってコマンドラインはあまり親しみのないものであるため、ADB を利用したバックアップもまた一般的な端末利用者にとって利用しやすいものではない。

2.3.4 パーソナルコンピュータと共に利用するアプリ

Helium[7] は clockworkmod 社が提供している Android 用バックアップツールである。Android 端末用とパーソナルコンピュータ用のそれぞれに用意された Helium アプリをインストールし、それらのアプリを通してバックアップを行う。Android 端末の GUI で操作ができ、フルバックアップが可能にもかかわらず Android 端末の root 化が不要なバックアップツールである。

利用の際に Android 端末の USB デバッグ機能を ON にしなければならない点やバックアップ時に表示される画面などから、おそらく前述した ADB のバックアップ機能に GUI をつけて利用しやすくしたものだと推測できる。

Helium を利用するためには、パーソナルコンピュータの利用が必要である。そのため、パーソナルコンピュータを所持していない端末利用者には利用できないことや、パーソナルコンピュータ側にもアプリ移行のツールのインストールや設定が必要であるという問題

点がある。

2.3.5 Google が提供する環境移行機能

Android5.0 から搭載された機能が Tap&Go である。この機能は、端末のセットアップ時に起動し、元々利用していた端末と Bluetooth でペアリングするか、NFC 同士をタップするだけで簡単に元の端末のデータを移行できるというものである。Android5.0 で追加された機能のため移行先端末の OS は Android5.0 でなければならないが、移行元端末の OS は Android5.0 以前でもかまわない。しかし当然のことながら、Android5.0 以前の端末同士ではこの機能は利用できない。

2.4 提案手法と既存ツールとの差分

本節では 2.3 節をふまえて、これらのツールと本研究の提案手法との差分について述べる。本研究では、移行ツールを Android アプリとして実装し、ADB や Helium のようにパーソナルコンピュータを利用せずに、移行元と移行先の Android 端末のみを利用してユーザの環境移行を実現することを目指す。このように Android 端末のみを利用する形にすることで、ユーザ環境移行のためにパーソナルコンピュータ等その他の端末を設定する手間を省くことができる。

root 化対象のアプリや Helium、Tap&Go といったツールを利用することによって Android 端末のユーザ環境を移行することは可能である。しかし、2.3 節で述べた通りそれぞれに何かしらの欠点が存在し、一般的な端末利用者にとって利用しやすいもの、手軽なものではない。そこで、本研究ではこれらの欠点を補った、端末利用者が手軽に扱えるツールを提案する。提案手法と既存ツールの比較を表 2.1 に示す。

アプリから端末中のデータを全て取得するためには管理者権限が必要である。しかし、2.3.2 節で述べた通り、管理者権限を取得するためには端末を root 化する必要がある。その一方で、端末を root 化する手順は複雑であったり、セキュリティが甘くなる等、一般的な端末利用者にとって好ましいものではない。そこで本研究では、Android の機能として管理者権限を利用できるように Android の管理者権限機構の改良を実装する。こうすることによってまず、端末利用者が端末を root 化する手間を削減することができる。

そして、root 化がセキュリティを甘くする理由として管理者権限を取得できるユーザに制限がないこと、管理者権限を取得できる期間に制限がないことが挙げられる。つまり、管理者権限を取得する事自体は端末上のユーザであれば誰からでもでき、管理者権限を取得してから終了を宣言するまで時間的な制約がなく管理者ユーザであり続けられるということである。そこで本研究では、管理者権限を取得できるユーザと管理者権限を取得でき

る時間を制限することによって、端末のセキュリティの低下をできるかぎり避けることにした。

また、Helium ではパーソナルコンピュータを利用しなければならず、Tap&Go では、ネットワーク通信が前提である。本提案ツールでは、これらの問題を Bluetooth 通信を利用することによって解決する。Bluetooth 通信を用いることで、環境移行を行う端末のみで完結するシステムを提案する。システム設計の詳細は 4 章で議論する。

表 2.1　既存ツールと提案システムの比較

	提案手法	一般的なアプリ	root 化アプリ	Helium	Tap&Go
環境移行が可能か	○	×	○	○	○
携帯端末以外が必要か	○	○	○	×	○
管理者権限が必要か	×	○	×	○	○
最新 OS でなくても利用可	○	○	○	○	×

第 3 章

Android の構造と管理者権限機構

3.1 Android の構造

本節では、本研究で扱う Android におけるセキュリティに関する構造について議論する。Android のアーキテクチャでは、Linux カーネルの上で標準ライブラリや Android ランタイム等が動作している。Linux カーネルを基にしていることから、Android は様々な概念や仕組みを Linux から継承している。

3.1.1 Android のセキュリティ構造

Android のセキュリティ構造は主に以下で説明する 4 つの要素によって成り立っている。

サンドボックス

Android アプリは、図 3.1 に示したように、それぞれがサンドボックスによって独立した状態で実行され、それぞれのサンドボックスには固有の Linux ユーザ ID(以下 UID) が割り当てられる。このようにすることによって、プロセス空間やメモリ空間を独立させ、アプリ間の干渉を防ぐことができる。また、アプリ同士が独立しているので、メッセージやデータのやり取りも普通にはできないようになっている。そのため Android には、アプリ間で通信を行うための仕組みが別途用意されている。詳しくは、3.1.2 節で説明する。

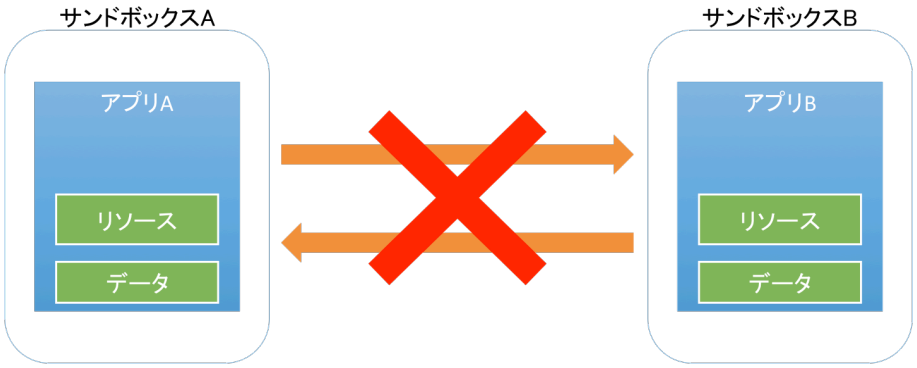

図 3.1　サンドボックス

パーミッション

パーミッションとは、Android 端末の一部のデータや機能をアプリ側が利用することを制御するための機能である。アプリから電話帳データやカメラ等の特定の機能へアクセスするためには、アプリのマニフェストファイルと呼ばれる xml ファイルに利用したいパーミッションを宣言しなければならない。マニフェストファイルへのパーミッション宣言の例をソースコード 3.1 に示す。

```
<!-- カメラの利用を許可するパーミッション　-->
<uses-permission android:name="android.permission.CAMERA">
<!-- 電話帳の読み込みを許可するパーミッション -->
<uses-permission android:name="android.permission.READ_CONTACTS">
```

ソースコード 3.1　AndroidManifest.xml におけるパーミッション宣言の例

マニフェストに宣言されたパーミッションは、図 3.2 や図 3.3 に示したように、そのアプリが利用する機能の一覧としてまとめられる。そしてこの一覧は、そのアプリのインストール時に端末利用者に提示され、端末利用者からの利用の承認を求める。端末利用者は、この一覧を見て、インストール中のアプリが利用する機能として相応しいかどうか、アプリがこの機能を利用しても構わないかを判断する。一覧表示されたパーミッションの全てを承認する場合には、アプリのインストールをインストールすることができる。しかし、1 つでも承認できないパーミッションが存在する場合には、アプリのインストールを中断し、そのアプリの利用を諦めることになる。

パーミッションの承認はアプリのインストール時にのみ静的に決定でき、アプリ側から動的にパーミッションを変更することはできない。また、端末利用者が一部のパーミッションをインストール後に拒否するように変更することもできない。

パーミッションには保護レベルというものが存在し、それぞれのパーミッションに設定された保護レベルによって扱いが異なる。パーミッションの保護レベルそれぞれ詳細は以

下のとおりである。

normal　端末利用者やシステムにとってリスクの少ないパーミッションであることを示す。そのため、インストール画面でも「全て表示」を選択しない限りは画面に表示されない。このレベルのパーミッションには、アラームの設定を許可する SET_ALARM やネットワーク状態の取得を許可する ACCESS_NETWORK_STATE 等がある。

dangerous　端末利用者やシステムにとってリスクがあるパーミッションであることを表す。インストール画面に表示されるのはこのレベルのパーミッションである。このレベルのパーミッションの例としては、電話の発信を許可する CALL_PHONE やインターネットアクセスを許可する INTERNET 等がある。

signature　パーミッションの定義元アプリと同じ署名をもつアプリにのみ、使用を許可するパーミッションである。システムが 2 つのアプリの証明書を比較してパーミッションの承認を行うためインストール画面には表示されない。

signatureOrSystem　システムレベルで利用されるパーミッション。システムイメージもしくは、システムイメージ内にあるアプリと同じ署名がされているアプリにのみパーミッションの利用を許可する。

normal、dangerous レベルのパーミッションが一般的によく利用される。図 3.3 に示した、インストール時に一覧表示されるパーミッションはこの 2 種類のどちらかである。また、カメラや電話帳の読み書き等の Android がアプリに提供している機能のパーミッションもこの 2 つのどちらかである。

signature レベルは、パーミッションの定義元と利用するアプリの署名が一致しなければならないというものである。そのため、同じ開発者が開発したアプリ同士でデータベースを共有したり、データのやり取りをする場合等の連携を図る場合等に利用される。

signatureOrPermission レベルは、システムイメージもしくはシステムアプリと同じ署名をしているアプリからしか利用できないためサードパーティ製のアプリからは利用できない。端末開発会社が独自機能を実装したアプリを提供したい場合などに用いられる。

ここで扱っている Android アプリの署名については本節内で後述する。

アクセス権

Android のファイルとディレクトリは、Linux 同様アクセス権によって管理されている。所有者とグループとその他のユーザの 3 種類に分けて、そのファイルに関しての読み込み権 (r)、書き込み権 (w)、実行権 (x) の 3 種類のアクセス権をそれぞれのグループに対して許可するものを与える。以下は Android の内部に保存されているファイルのアクセス権を表示した例である。

図 3.2　GooglePlay でのパーミッションリストの表示

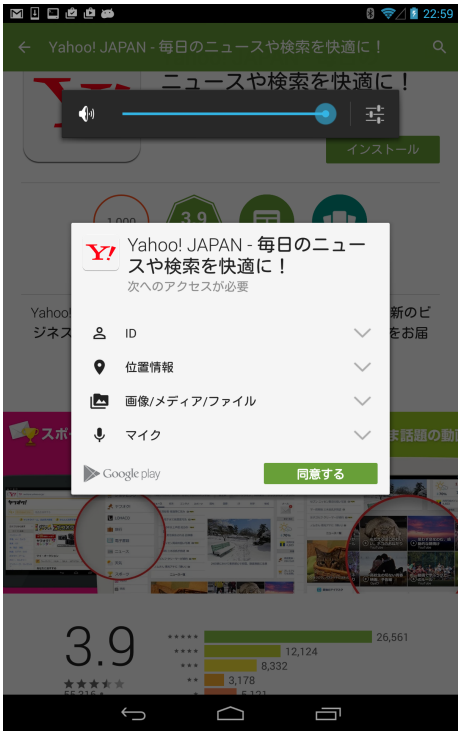

図 3.3　アプリインストール時のパーミッションリストの表示

```
shell@deb:/sdcard/Pictures/Screenshots $ ls -l
-rw-rw-r-- root      sdcard_rw    156762 2014-02-19 17:50 Screenshot.png
```

この例の `rw-rw-r--`と書かれている部分がファイルのアクセス権を表現している。上述したアクセス権 *rwx* のどれかが書かれているところはそのアクセス権が与えられていることを表現している。反対にアクセス権が与えられていないものは-で表現されている。また、その右側の root が所有者ユーザ、`sdcard_rw` が所有グループを表している。

従ってこの例の場合は、所有者・グループである root ユーザと `sdcard_rw` に属するユーザにはこのファイルの読み書きが許可されている。一方で、その他のユーザはこの

ファイルの読み込みは許可されているが、書き込みは許可されていない。

また、今回の例で見たファイルは実行ファイルではないため実行権は必要ないので全てのユーザに実行権は与えられていない。

署名と証明書

Android アプリには、開発元を証明する証明書を利用したデジタル署名が行われていなければならない。署名は Google Play への公開時や Android 端末へインストールするときに検証される。証明書は、認証局によって発行された証明書を利用することもできるが、一般的には Android アプリの開発キットに付属しているツールを利用して、開発者自らが発行する。そのため Android における証明書では、通常のデジタル署名が保証するようなファイル発信者の安全性ではなく、アプリの更新の発行元の整合性を保証する。つまり、初回インストール時にそのアプリが安全なものなのか、製作元は信用できるところなのかということは保証されない。しかし、アプリの更新時には発行元が必ず同じであるということは保証されるので、悪意のある開発者によってこのアプリが改竄されていないことが保証される。

3.1.2 Android のプロセス間通信

前節で説明したように、Android では基本的にアプリが端末内のデータにアクセスすることやアプリ同士でデータのやり取りができないようになっている。本節では、この問題を解決するために用意されている Android の仕組みについて説明する。

Intent

Android アプリは、アプリケーションコンポーネントと呼ばれるものの組み合わせで実装される。アプリケーションコンポーネントの例としては、アプリの画面を表現する Activity やアプリのバックグラウンドで処理を実行する Service が挙げられる。Intent とは、このようなコンポーネント間でメッセージをやりとりするための仕組みのことである。例えば、とある Activity から別の Activity を起動して画面遷移を実現したり、Service での処理中に必要な Activity を起動させることが Intent によって可能である。実際には一度 Android システムが仲介に入り Intent の送信先を探すように実装されている。

Intent には、宛先を明確に指定する明示的 Intent とアクションのみを指定し実際の宛先はシステムや環境によって決定される暗黙的 Intent がある。

明示的 Intent の例を図 3.4 に示す。明示的 Intent は、Intent の宛先となるコンポーネント名を指定して発行されるため、宛先のコンポーネント名を知らなければならない。しかし一般的にアプリ内部のコンポーネント名は公開されないので、他のアプリとの連携な

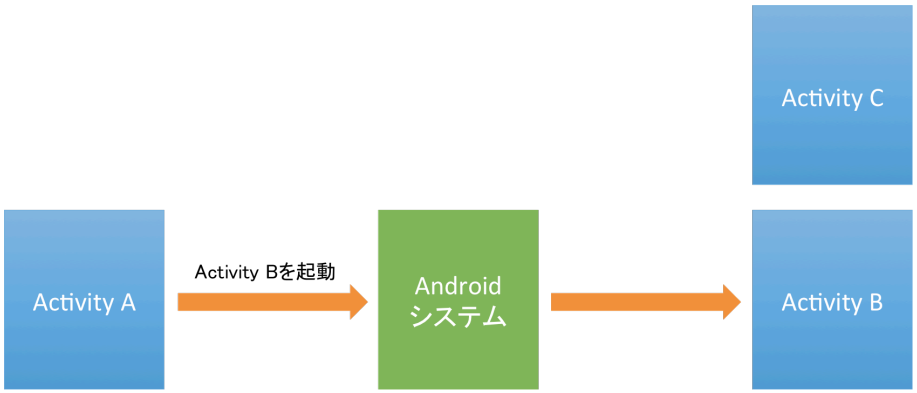

図 3.4　明示的 Intent の例

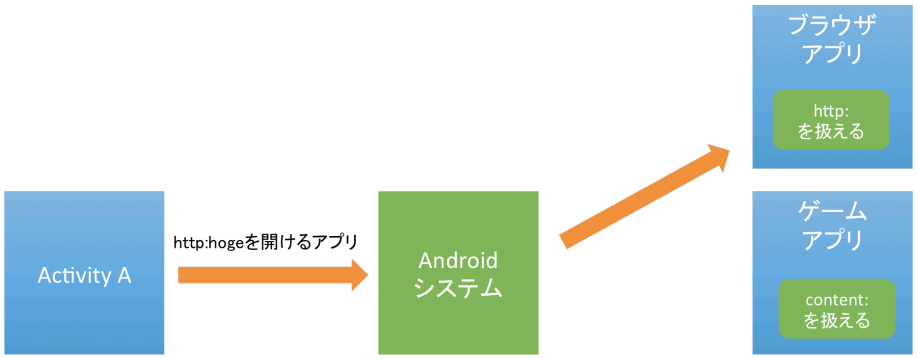

図 3.5　暗黙的 Intent の例

どでは利用されずアプリ内部の画面遷移等でよく利用される。

暗黙的 Intent は、図 3.5 に示した例のように、ターゲットのコンポーネント名を指定せず、実行したい所定の動作だけを指定する。また各アプリは、自らが処理可能な Intent を IntentFilter を利用することで指定できる。Intent の発行元からは特に指定をせずに、実行可能なアプリが指定された動作を実行するという緩い結びつきで連携を実現している。これは Android システムが受け取った Intent の動作を処理可能であるという IntentFilter をもったアプリを探し、そこに Intent を送ることで実現している。暗黙的 Intent は、コンポーネント名を知らなくても発行可能なため明示的 Intent では難しかった、ブラウザアプリを利用して Twitter 認証をする場合等、外部のアプリと連携する場合に利用される。

ContentProvider

ContentProvider は、アプリが持つデータを外部に向けて公開する仕組みである。3.1.1 節で説明した通り、アプリはそれぞれリソースが独立されており、他のアプリから情報を読み取ることや、他のアプリへ情報を渡す事ができないようになっている。しかし、

アプリによっては他のアプリが持っている情報を受け取って処理したり、またその逆にアプリの持つ情報を公開したい場合がある。これを解決するために用意されているのが ContentProvider である。例えば、電話帳アプリ等はこの仕組を用いて実装されており、電話帳データの読み込みや書き込みは ContentProvider を通して他のアプリから実行することができる。

3.2 root 化

root 化とは、Android 端末の管理者権限 (root 権限) を取得できる状態にすることを指す。3.1.1 節で説明したように、Android のファイルやディレクトリへのアクセスはアクセス権によって制御されている。そのため、一般ユーザからはアクセス出来ないファイルが多数存在し、実現できる操作が制限されていると言える。しかし、管理者権限を利用することによって全てのファイルやディレクトリへのアクセスが可能となるため、システムで利用しているフォントの入れ替えから、OS イメージの入れ替えなど様々なことができるようになる。

市販の Android 端末は、端末利用者が誤ってシステムにとって重要なファイルの書き換えや削除を行わないようにするため、端末利用者側から管理者権限を利用する手段を一切用意していない。一般的な Linux では、*su* や *sudo* コマンドを利用することで管理者権限を利用することができるが、Android では、このようなコマンドを一切端末利用者に提供していない。

そこで、AndroidOS の脆弱性をついて一時的に管理者権限を奪取し、端末内に *su* コマンドを一般ユーザから実行可能な場所、アクセス権で設置することで、それ以降も管理者権限を一般ユーザから利用できるようにするというのが root 化と呼ばれる手法である。

第 4 章

提案システムの設計

4.1 設計全貌

本研究では、ユーザ環境の移行を実現するために、管理者権限機構の改良とデータ移行ツールの開発を行う。これらはそれぞれ Android のアプリとして実装する。提案するシステムの全体像を図 4.1 に示す。

提案システムでは、管理者権限機構の改良実装である AndroidSudoManager とデータ移行ツールの実装である EnvironmentMigrator の 2 つを 1 つの端末上にインストールする。AndroidSudoManager では、受け取ったコマンドを管理者として実行し、必要であれば実行結果を EnvironmentMigrator へ返す。EnvironmentMigrator では、ユーザインタフェースの提供、移行データの収集・配置、端末間通信を行う。データの収集・配置の過程で管理者権限を利用する必要がある際には、AndroidSudoManager に実行したい Linux コマンドを送信する。

それぞれの詳細な設計については次節以降で解説する。

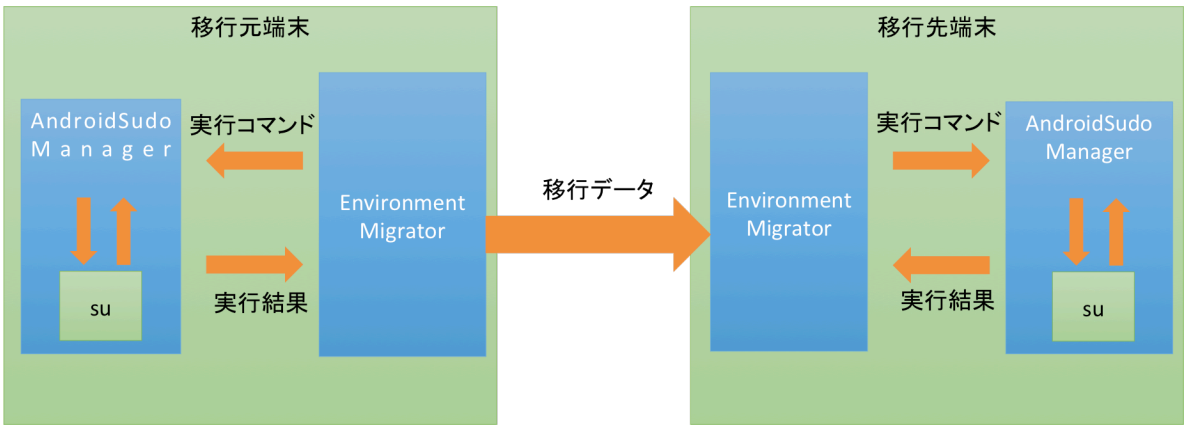

図 4.1 提案システム全体図

4.1.1 管理者権限機構の改良

3 章で説明した通り、Android の端末内のデータはシステムの仕組みによってアクセスが制御されている。また、OS の管理者権限についてもセキュリティ等を理由に端末利用者が利用できる形で用意されていない。そこで、本研究では端末のセキュリティを考慮した上で、アプリに管理者権限を利用する機能を提供できるツール AndroidSudoManager を提案する。AndroidSudoManager は、システムの一部とみなすため AndroidOS の構築時にインストールされるプリインストールアプリとして実装する。AndroidSudoManager に求める要求条件は以下のとおりである。

1. 管理者権限を行使することができる
2. 管理者権限の利用を最小限に制御できる

1 点目の条件を解決するために、本研究では *su* コマンドを用いる。ただし、2 点目の条件を満たすために *su* コマンドを配置する場所を変更する。具体的には、AndroidSudoManager のデータディレクトリ (/data/data/androidsudomanager) の下にディレクトリ (/data/data/androidsudomanager/mysu) を作成し su コマンドを配置する。そして、このディレクトリ (mysu) のアクセス権を所有者のみアクセス可能な状態に設定する。こうすることによって、*su* コマンドの実行できるユーザが AndroidSudoManager のみとなる。つまり、管理者権限を利用するためには AndroidSudoManager を利用しなければならない。

次に、AndroidSudoManager へのアクセスをパーミッションを用いて制限する。Android のパーミッションには保護レベルがあり、そのうちの *signature* を指定することによって、同じ鍵で署名されたアプリのみにアクセスを許すように設定する。利用アプリが AndroidSudoManager と同じ鍵で署名されていなければならないため、外部からインストールされたアプリは AndroidSudoManager を利用することができない。こうすることによって、管理者権限を扱えるアプリを制限することができる。

また、AndroidSudoManager 内部での *su* コマンドの実行は必ず以下のように`-c` オプション付きで行う。

$su -c *command*

`-c` オプションを指定することによって、管理者に切り替え後に *command* を実行する。実行後は管理者ではなく実行ユーザに戻るため、管理者になる時間をコマンド実行中のみに限定することができる。

本提案ツールを EnvrionmentMigrator と AndroidSudoManager の 2 つのアプリに分け、EnvironmentMigrator 内部に管理者権限機構を組み込まなかった理由は拡張性と安全性を考えてのことある。管理者権限を制御する機能を分離しておくことによって、今後データ移行以外の用途で管理者権限の利用が必要となった際に利用することが容易となる。また、アプリが分離していない場合には、データ移行ツールである EnvironmentMigrator を改竄してしまえば悪用が可能である。しかし、本手法のようにアプリを分離し *signature* パーミッションによって保護することによって、安全性を向上させることが可能である。

以上により、端末の安全性を極力保ったまま管理者権限を利用できるように改良する。

4.1.2 移行ツールの設計

本節では、環境移行を行うアプリ EnvironmentMigrator の設計について説明する。EnvrionmentMigrator の構成図を図 4.2 に示す。EnvironmentMigrator では、ユーザインタフェースを提供する UserActivity モジュール、データの収集と配置を行う DataManager、端末間の通信を行う BluetoothManager を用意する。

図 4.3 にデータ移行元の、 図 4.4 にデータ移行先の本アプリのフロー図を示す。移行元では、データの収集を行い送信する。図 4.3 に示したように、アプリを起動すると端末中のデータの収集を開始する。Android フレームワークの API や ContetProvider を利用して取得できるデータについては、それを利用する。Android フレームワークで提供されている方法で取得不可能な場合には、独自の手法を用いてデータを収集する。その際に管理者権限が必要な場合には、AndroidSudoManager へとコマンドを発行する。

AndroidSudoManager へのコマンドの発行は、Intent を利用して実装する。AndroidSudoManager への明示的 Intent を発行し、その Intet にコマンドを文字列データとして送信する。AndroidSudoManager では、Intent の中身を取り出し文字列データとして受け取ったコマンドを管理者として実行し、必要であれば結果を返送する。

移行先では、受信したデータを適切な形で端末中に配置する。図 4.4 に示した通り、移行先ではデータを受信し、端末の適切な位置へとそのデータを配置する。その際に、Android フレームワークの API を利用して書き込める場合にはそれを利用してデータを配置する。そうでない場合には、適切な位置へとファイルを配置する。その際に管理者権限が必要な場合には、AndroidSudoManager へと実行するコマンドを発行して処理させることでデータを配置する。受信したデータを全て配置し終えたら、受信したデータが不正に利用されることがないように不要なデータは削除する。削除が完了したらアプリを終了させ、移行が完了する。

本アプリでは、端末間の通信に Bluetooth 規格を用いる。これは、本アプリが端末の買

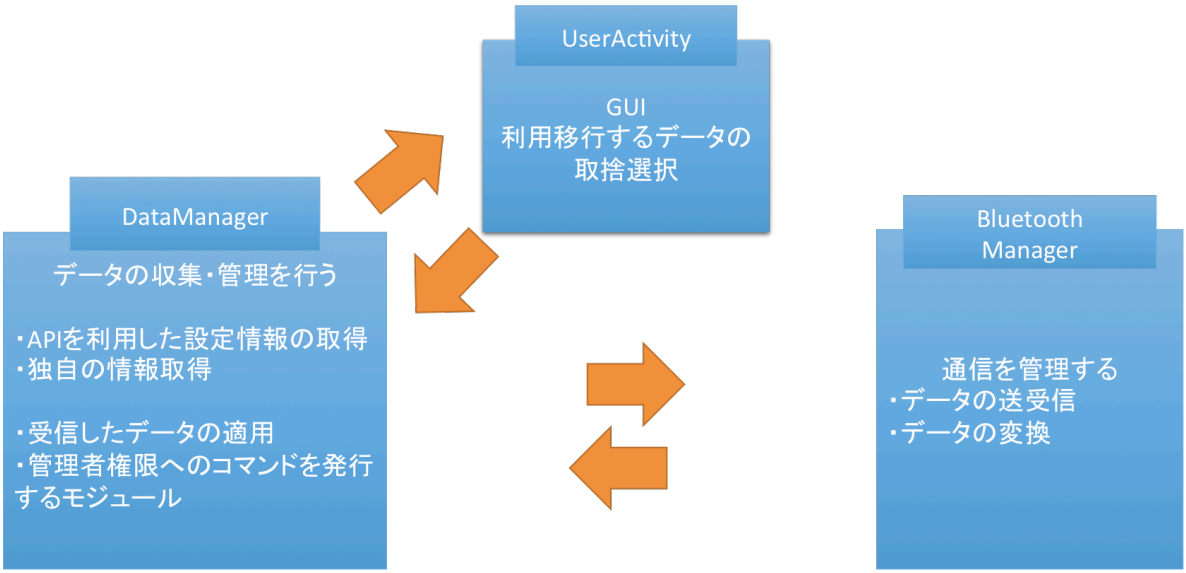

図 4.2　EnvironmentMigrator モジュール設計図

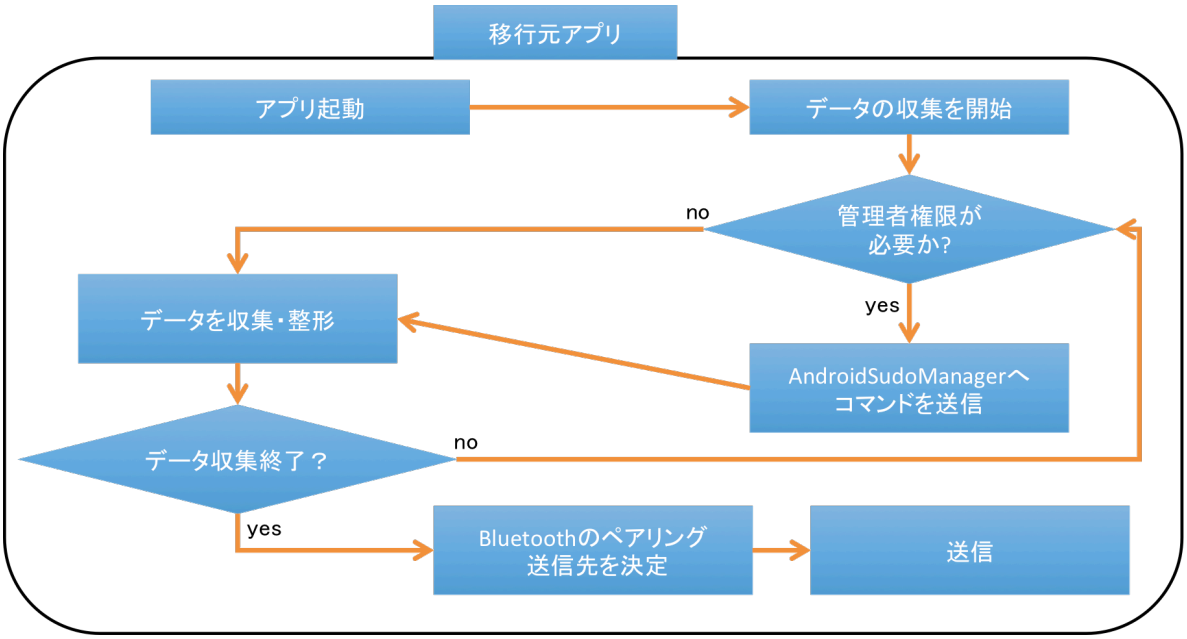

図 4.3　データ移行元 EnvironmentMigrator フロー図

い替え時での使用を想定しており、端末の買い替えた直後の場合、古い方の端末は 3G や 4G の通信手段を持たないためである。近年のスマートフォンには、必ずと言っていいほど Bluetooth が搭載されており、通信契約などが必要ないためその場で手軽に通信できる手段として本研究では採用した。

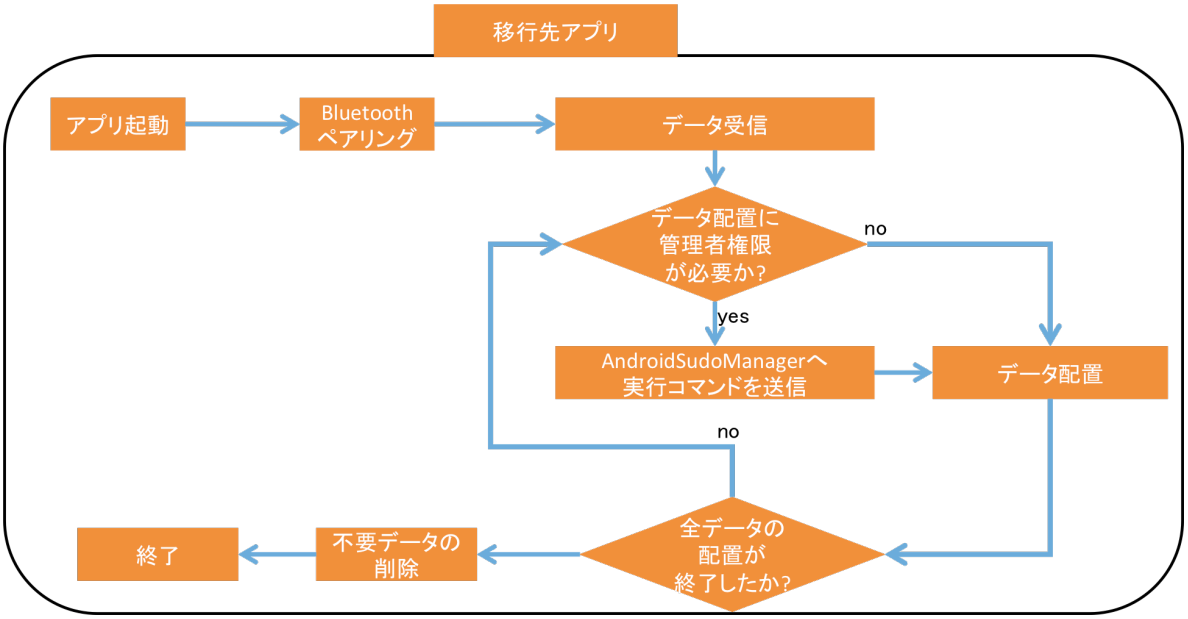

図 4.4　データ移行先 EnvironmentMigrator フロー図

第 5 章

提案システムの実装

5.1 管理者権限機構の改良

本節では、管理者権限機構の改良である AndroidSudoManager の実装について議論する。

AndroidSudoManager は、ソースコード 5.1 に示すように、Java の RunTime クラスを用いて Linux コマンドをアプリから実行できるように実装している。

```
1 Runtime runtime = Runtime.getRuntime();
2 Process process = runtime.exec(command);
```

ソースコード 5.1 Runtime クラス Linux コマンドの実行

5.2.2 節で紹介するデータ移行ツールとの連携には、暗黙的 Intent を利用している。データ移行ツールからの暗黙的 Intent を利用したコマンド送信の実装をソースコード 5.2 に示す。暗黙的 Intent では、宛先の指定は行わずに setAction メソッドを利用して Intent の受け取り側が実行する処理を指定する。そして、指定された Action を受け取るように IntentFilter が書かれたアプリのみが Intent を受け取る。本研究実装では、他のアプリがこの Intent を受け取らないようにするために、Intent クラスに用意された共通 Action ではなく独自で jp.ac.uec.is.spa.hiromu.SUDO という Action を定義した。そして、putExtra メソッドでテキストデータとしてコマンドを Intent に載せて、startActivity メソッドで AndroidSudoManager の Activity をスタートさせる。

```
1   private static final String INTENT_SUDO = "jp.ac.uec.is.spa.hiromu.
        SUDO";
2
3   public static void sendRequest(Context context, String command){
4        Intent intent = new Intent();
5        intent.setAction(INTENT_SUDO);
6        intent.setType("text/plain");
7        intent.putExtra(Intent.EXTRA_TEXT, command);
8        context.startActivity(intent);
9    }
```

ソースコード 5.2 データ移行ツールからのコマンド送信方法

Intent を受け取るための IntentFileter の記述をソースコード 5.3 に示す。action タグで受け取る Action を指定している。今回は独自で定義した jp.ac.uec.is.spa.hiromu.SUDO を指定している。

```
<intent-filter>
    <action android:name="jp.ac.uec.is.spa.hiromu.SUDO" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/*" />
</intent-filter>
```

ソースコード 5.3　IntentFileter の記述

Intent を受け取った AndroidSudoManager は、Intent から実行するコマンドの文字列を取り出す。受け取った Intent からデータを取り出す方法についてソースコード 5.4 に示す。getIntent メソッドを利用することで Intent を取得する。取得した Intent の Action を getAction メソッドで取り出す。取り出した Action が jp.ac.uec.is.spa.hiromu.SUDO であった場合には、Intent から文字列データを取り出してコマンド実行を行う。

```
    /**Intent を取得*/
    Intent intent = getIntent();
    /**Intent の Action を取得*/
    String action = intent.getAction();
    /**Action が自分で定義した SUDO だったら*/
    if (INTENT_SUDO.equals(action)) {
        /**Intent からデータを取り出す*/
        Bundle extras = intent.getExtras();
        /**データが空だったら中止*/
        if (extras == null) return;
        /**データが入っていたら文字列を取り出す*/
        CharSequence ext = extras.getCharSequence(Intent.EXTRA_TEXT);
        /**データが文字列でなかったら中止*/
        if (ext == null) return;
        /**受け取った文字列をスペース区切りでスプリット*/
        String[] command = ((String)ext).split("␣");
        /**コマンドの実行*/
        CommandLineExecuter.execCommand(command,true);
   }
```

ソースコード 5.4　Intent からコマンド文字列の取り出し

実際にコマンドの実行を行うコードをソースコード 5.5 に示す。提案手法の方針として管理者権限を利用するのはコマンドの実行時のみに限定するために、恒久的に管理者権限になるようなコマンドは受け付けないようにしたい。そこで、コマンドを実行する前に文字列を検査する sucheck メソッドを用意した。sucheck メソッドでは引数として与えられたコマンド中に *su* 及び *sh* が含まれていないか検査をする。*sh* をチェックすることによって、*su* を含むシェルスクリプトが実行されることを回避する。これらの文字列がコマンド中で見つかった場合には Exception を投げてエラーとして扱う実装になっている。

与えられたコマンドに *su* が含まれないことを確認できたら、*su -c* をコマンドの先頭に追加する setSudo メソッドを用意しているのでこれを実行する。

checksu メソッドと setSudo メソッドによって、実行されるコマンドが以下の形であることが保証できる。

```
$su -c command
```

また command 中に *su* が含まれないことを保証できる。したがって、本手法においては管理者権限の施行に関して期間的な制限を設けることができる。

加えて、signature レベルのパーミッションの定義する事によって、このアプリを呼び出す事ができるアプリを制限する。AndroidSudoManager に定義したパーミッションをソースコード 5.6 に示す。

name 属性に書かれている *jp.ac.uec.is.spa.hiromu.androidsudomanager.permission.SUDO* がパーミッション名である。端末内に同じパーミッションが複数存在すると開発者が想定していない誤動作が起こりえることから、通例として Package-Name.permission.PermissionName という形式で宣言することが推奨されている。これは、Java ではパッケージ名を世界中で一意にするために、開発者のもつ Web ページのドメインを逆から読んだものにアプリケーション名をつけてパッケージ名とすることに習ったものである。

protectionLevel 属性によってこのパーミッションの保護レベルを指定することができる。signature レベルを指定することによって、パーミッションを定義したアプリと同じ鍵で署名されていなければ利用することができない。よってこれを宣言することによって管理者権限機構の利用を行えるアプリを制限できると考えられる。

```
1  /**
2   * アプリからコマンド実行を行うクラス
3   */
4  public class CommandLineExecuter {
5      /**コマンドを実行するメソッド
6       * @param command 指定されたコマンド
7       * */
8      public static String execCommand(String[] command, boolean sudo){
9          /**コマンドにsuが含まれていないかチェックする*/
10         sucheck(command);
11         /**コマンドにsuを設定する*/
12         if(sudo){
13             command=setSudo(command);
14         }
15         Runtime runtime = Runtime.getRuntime();
16         Process process;
17         StringBuffer output = new StringBuffer();
18         String line = "";
19         BufferedReader reader;
20         BufferedReader errReader;
21         try{
22             /**コマンドを実行*/
23             process = runtime.exec(command);
24             /**実行結果の入ったバッファを取得*/
25             reader = new BufferedReader(
26                         new InputStreamReader(process.getInputStream())
27             );
28             /**エラーの入ったバッファを取得*/
29             errReader = new BufferedReader(new InputStreamReader(
                   process.getErrorStream()));
30             /**結果を一行ずつ追加*/
31             while((line = reader.readLine()) != null){
32                 output.append(line);
33                 output.append("\n");
34             }
35
36             /**errorのバッファも処理する*/
37             output.append("error:\n");
38
39             while((line = errReader.readLine())!= null){
40                 output.append(line);
41                 output.append(("\n"));
42             }
43             reader.close();
44             errReader.close();
45             process.waitFor();
46
47         }catch(IOException ioe){
```

```
48              ioe.printStackTrace();
49              return ioe.toString();
50          }catch(InterruptedException itre){
51              itre.printStackTrace();
52              return itre.toString();
53          }
54          /**実行結果を返す*/
55          return output.toString();
56      }
57      /**渡されたコマンドにsuコマンドが含まれていないかチェックする */
58      public static void sucheck(String[] command){
59          for(int i=0; i<command.length; i++){
60              if(command[i].equals("su")){
61                  throw new IllegalArgumentException("this␣command␣is␣
                        including␣'su'!!");
62              }
63          }
64      }
65      /**コマンドの先頭にsu -c を追加する*/
66      public static String[] setSudo(String[] command){
67          List<String> tmpList = new ArrayList<String>();
68          tmpList.add("su");
69          tmpList.add("-c");
70          tmpList.addAll(Arrays.asList(command));
71          Log.d("setSudo",tmpList.toString());
72          String[] sudo_command = (String[])(tmpList.toArray(new String[
                tmpList.size()]));
73          return sudo_command;
74      }
75  }
```

ソースコード 5.5　CommandExecuter.java

```
1       <!-- パーミッションの宣言 -->
2       <permission
3        android:name="jp.ac.uec.is.spa.hiromu.androidsudomanager.
             permission.SUDO"
4        android:protectionLevel="signature"
5        android:label="@string/permlab_sudo"
6        android:description="@string/permdesc_sudo"
7        android:permissionGroup="android.permission-group.SYSTEM_TOOLS"/>
```

ソースコード 5.6　パーミッションの定義と宣言

5.2 データ移行ツール

本節ではデータ移行ツールである EnvironmentMigrator の実装方法について紹介する。EnvironmentMigrator は、BluetoothManger、DataManager、UserActivity の 3 つからなる。UserActivity は利用者にフィードバックを行う GUI であるが、これについては実装時の確認程度にしか実装していないため、今回は議論の対象として扱わない。

5.2.1 BluetoothManager

BluetoothManager では、接続可能デバイスの検索、Bluetooth ペアリング、通信処理等の Bluetooth 通信に関わる部分を実装している。

初めに、利用している端末が Bluetooth をサポートしているのかを調べる。ソースコード 5.7 にコード例を示す。端末が Bluetoot をサポートしているか調べるには、BluetoothAdapter を取得する getDefaultAdapter メソッドを利用する。getDefaultAdapter メソッドは、BluetoothAdapter のインスタンスを取得するものであるが、端末が Bluetooth に対応していない場合には null を返すようになっている。そこで BluetoothSupportCheck() では、getDefaultAdapter メソッドの返り値が null かどうかチェックして boolean を返すようにしている。

また、isEnabled メソッドを利用する事によって現在端末の Bluetooth が有効になっているかどうかが取得できる。

```
1   /**
2    * 端末がBluetooth に対応しているかをチェックする true →対応，
         false →非対応
3    */
4   public boolean BluetoothSupportCheck() {
5       return BluetoothAdapter.getDefaultAdapter() != null;
6   }
7   /** Bluetooth が有効になっているかを返す */
8   public boolean isEnabled() {
9       return mBtAdapter.isEnabled();
10  }
```

ソースコード 5.7　利用端末の Bluetooth をサポートしているか調べる

ソースコード 5.7 で示したメソッドを実際に利用している部分をソースコード 5.8 に示す。まず、BluetoothSupportCheck メソッドで Bluetooth がサポートされている端末かどうかをチェックする。もしサポートされていなかった場合には、finish メソッドを使ってアプリを終了するようになっている。サポートされていた場合には、Bluetooth が有効になっているかをチェックする。Bluetooth が無効になっていた場合

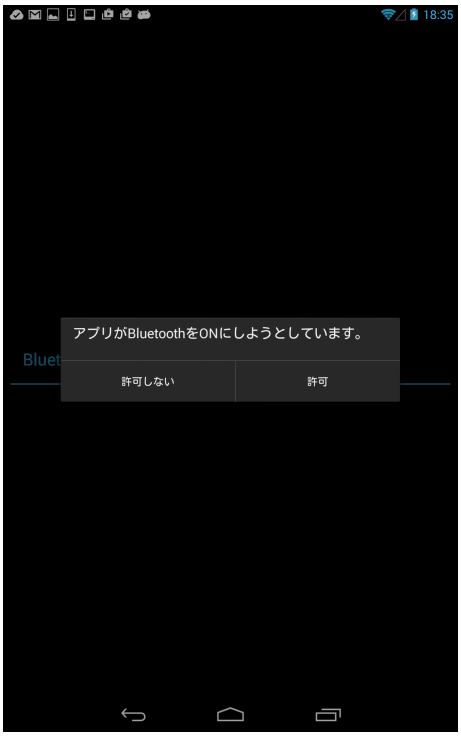

図 5.1　Bluetooth 有効に求める画面

には、ACTION_REQUEST_ENABLE を実行できるアプリに暗黙的 Intent を発行する。この Action は、図 5.1 のように、ユーザに Bluetooth を有効にするように求めるダイアログを表示する。

```
1        /** 端末がBluetooth に対応していなかったらここで終了 */
2        if (!mBtManager.BluetoothSupportCheck())
3            finish();
4        /** Bluetooth が有効になっていなければ有効にするように促す */
5        if (!mBtManager.isEnabled()) {
6            Intent requestBtOn = new Intent(BluetoothAdapter.
                 ACTION_REQUEST_ENABLE);
7            startActivityForResult(requestBtOn, BluetoothManager.
                 REQUEST_ENABLE_BLUETOOTH);
8        }
```

ソースコード 5.8　Bluetooth 利用の事前準備コード例

StartActivityForResult メソッドで開始した Activity からは値を返させることができる。返された値を受け取り処理するためには、ソースコード 5.9 示したように、Intent を発行した Acitivity で onActivityResult メソッドをオーバライドして実装する。引数の requestCode には要求した Action、ResultCode には返却された値が格納されいている。ソースコード 5.9 は、本アプリで実際に使われているコードで、ここでは requestCode が Bluetooth を有効にすることを求めた先から値が返ってきているかをチェックしている。

requestCode が正しかった場合には、Bluetooth が有効にされたかどうかを ResultCode で判断している。ResultCode から、Bluetooth が有効にされたことがわかった場合には Log を出力し、有効にされなかった場合にはアプリを終了するようになっている。

```
1    @Override
2    protected void onActivityResult(int requestCode, int ResultCode,
         Intent data) {
3        if (requestCode == BluetoothManager.REQUEST_ENABLE_BLUETOOTH) {
4            if (ResultCode == Activity.RESULT_OK) {
5                Log.d("Bluetooth.Switch", "ON");
6            } else {
7                finish();
8            }
9        }
10   }
```

ソースコード 5.9　Intent 発行した先から返り値を受け取る

接続可能デバイスを検索する方法の例をソースコード 5.10 に示す。ここではまず、Broadcast される Intent の内の必要なものだけを受け取るための IntentFilter を用意している。Broadcast される Intent を受け取るためにこの IntentFilter と共に BroadcastReceiver のオブジェクトを registerReceiver メソッドでシステムに登録している。実際に Broadcast を受け取った際の処理をソースコード 5.11 に記述している。今回は、接続可能なデバイスを発見したら画面のリストに詰めるように実装している。したがって、デバイスが発見された ACTION_FOUND の時とデバイス名が判明した ACTION_NAME_CHANGE の時には画面のリストを更新し、探索の開始 ACTION_DISCOVERY_STARTED と終了 ACTION_DISCOVERY_FINISHED の時には、通信状況を把握するためにデバッグログを出力させている。

Bluetooth のデバイス検索は、システムと大量の通信を発生させるためバッテリーの消耗などが激しい。加えて、二重に検索をかけてしまうとリストの整合性を保てなくなる。そのため、検索を開始する前に今現在デバイスの検索が行われているのかをチェックし、行われている場合にはその検索をキャンセルしてから、端末の検索を開始するように実装している。

```
1                /**検索状態を受け取るためのフィルタを用意*/
2                IntentFilter filter = new IntentFilter();
3                filter.addAction(BluetoothAdapter.
                     ACTION_DISCOVERY_STARTED);
4                filter.addAction(BluetoothAdapter.
                     ACTION_DISCOVERY_FINISHED);
5                filter.addAction(BluetoothDevice.ACTION_NAME_CHANGED);
6                filter.addAction(BluetoothDevice.ACTION_FOUND);
7                registerReceiver(DeviceFoundReceiver, filter);
```

```
8                 /** 既に検索中の場合は検索をキャンセルする */
9                 if (mBtManager.isDiscovering()) {
10                    mBtManager.cancelDiscovery();
11                }
12                /**ペアリングされているデバイスのリストを取得*/
13                Set<BluetoothDevice> pairedDevices = mBtManager.
                      getBoundedDevices();
14                /** 接続可能端末を検索開始 */
15                mBtManager.startDiscovery();
```

ソースコード 5.10　Bluetooth 接続可能なデバイスの検索方法

```
1  private final BroadcastReceiver DeviceFoundReceiver = new
       BroadcastReceiver() {
2          @Override
3          public void onReceive(final Context context, Intent intent) {
4              String action = intent.getAction();
5              String dName = null;
6              BluetoothDevice foundDevice;
7              ListView nonpairedList = (ListView) findViewById(R.id.
                   nonpairedDeviceList);
8              nonpairedList.setOnItemClickListener(((DeviceListActivity)
                   context));
9              /**端末の検索が開始された*/
10             if (BluetoothAdapter.ACTION_DISCOVERY_STARTED.equals(action
                   )) {
11                 Log.d("start", "scan started!!!");
12             }
13             /**端末が見つかった*/
14             if (BluetoothDevice.ACTION_FOUND.equals(action)) {
15                 foundDevice = intent.getParcelableExtra(BluetoothDevice.
                       EXTRA_DEVICE);
16                 foundDeviceList.add(foundDevice);
17                 dName = foundDevice.getName();
18                 if (dName == null)
19                     return;
20                 if (foundDevice.getBondState() != BluetoothDevice.
                       BOND_BONDED) {
21                     pairedDeviceAdapter.add(dName + "¥n" + foundDevice.
                           getAddress());
22                     Log.d("action found", dName);
23                 }
24                 nonpairedList.setAdapter(pairedDeviceAdapter);
25             }
26             /**端末名が判明した*/
27             if (BluetoothDevice.ACTION_NAME_CHANGED.equals(action)) {
28                 foundDevice = intent.getParcelableExtra(BluetoothDevice.
                       EXTRA_DEVICE);
29                 foundDeviceList.add(foundDevice);
```

```
30                if (foundDevice.getBondState() != BluetoothDevice.
                      BOND_BONDED) {
31                    pairedDeviceAdapter.add(dName + "¥n" + foundDevice.
                          getAddress());
32                }
33                nonpairedList.setAdapter(pairedDeviceAdapter);
34            }
35            /**端末検索が終了した*/
36            if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED.equals(
                  action)) {
37                Log.d("finished", "scan␣finished!!!");
38            }
39        }
40    };
```

ソースコード 5.11　BroadcastReceiver

Bluetooth の通信では、ソケット通信が用いられる。本実装ではサーバクライアント方式で通信を行うように実装した。サーバサイドでは listenUsingRfcommWithServiceRecord を用いて、自らの BluetoothAdapter からソケットを作成する。ソケットを作成したサーバは、メインスレッドとは別にスレッドで accept メソッドを実行し、クライアントからの接続を待つ。ソースコード 5.12 に実装したコードの一部を示す。

```
1  /**サーバソケットの作成*/
2          …
3          BluetoothServerSocket tmpServerSock = null;
4          mServerAdapter = btAdapter;
5          try {
6              tmpSock = device.createInsecureRfcommSocketToServiceRecord(
                   BLUETOOTH_CLIENT_UUID);
7          } catch (IOException e) {
8              e.printStackTrace();
9          }
10         …
11 /**スレッドがスタートしたら実行される処理*/
12     public void run() {
13         BluetoothSocket receivedSocket = null;
14         while (running) {
15                 receivedSocket = serverSock.accept();
16             if (receivedSocket != null && receivedSocket.isConnected())
                    {
17                 /**実際に書き込みを行う処理*/
18             }
19         }
20             serverSock.close();
21     }
```

ソースコード 5.12　サーバスレッドのコード例の一部

クライアントサイドでは、接続先の Bluetooth デバイスを抽象化した BluetoothDevice クラスのインスタンスから、createInsecureRfcommSocketToServiceRecord メソッドを利用してクライアントソケットを作成する。作成したソケットから connect メソッドを利用してサーバソケットと通信路を確立する。同じくクライアントサイドも実装したコードの一部をソースコード 5.13 に示す。

```
/**クライアントソケットの作成*/
        …
        BluetoothSocket tmpSock = null;
        mDevice = device;
        try {
            tmpSock = device.createInsecureRfcommSocketToServiceRecord(
                BLUETOOTH_CLIENT_UUID);
        } catch (IOException e) {
            e.printStackTrace();
        }
        …
 /**スレッドがスタートしたら実行される処理*/
    public void run() {
        /**まだ接続する相手を探していたら検索を終了させる*/
        if (myClientAdapter.isDiscovering()) {
            myClientAdapter.cancelDiscovery();
        }
        try {
            /**ソケットで接続*/
            clientSocket.connect();
            Log.d("client␣connect", "connect");
        } catch (IOException e) {
            try {
                /**エラーがあった場合にはソケットを閉じて終了する*/
                clientSocket.close();
            } catch (IOException closeException) {
                e.printStackTrace();
                closeException.printStackTrace();
            }
            return;
        }
        /**実際には書き込み処理を行う*/
    }
```

ソースコード 5.13　クライアントサイドのコードの一部

実際に文字列やファイルを通信で送るためには、ソケットのインスタンスから Java の入出力に利用される概念である Stream を取得して行う。実装したコードの一部をソースコード 5.14 に示す。OutputStream に書き込むことによってデータを送信する事ができ、InputStream から読み込むことによってデータを受信することができる。本研究中で

は、ブラウザブックマークと無線 LAN 通信は JSONObject を String に変換して送受信している。一方で、アプリケーションパッケージ等のファイルは、読み込んだファイルを Serializable オブジェクトとして扱い送受信している。受け取ったオブジェクトは byte 列を順番に FileOutputStream に書き出すことによって復元される。

```
…
/**ソケットから Stream を取得する*/
        try {
            in = socket.getInputStream();
            out = socket.getOutputStream();
        } catch (IOException e) {
            e.printStackTrace();
        }
…
/**書き込みを行う*/
        try {
            write(sendMessage.getBytes("UTF-8"));
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
…
/**読み込みを行う*/
        try {
            tmpBuf = in.read(buff);
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (tmpBuf != 0) {
            try {
                /**読み込んだバッファを String に変換する*/
                rcvmsg = new String(buff, "UTF-8");
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }
        }
…
   /**
     * Serializable な Object を書き込む
     */
    public void write(Serializable obj) {
        try {
            ObjectOutputStream oout = new ObjectOutputStream(new
                BufferedOutputStream(out));
            oout.writeObject(obj);
            oout.flush();
            oout.close();
        } catch (IOException e) {
            e.printStackTrace();
```

```
43          }
44      }
45  …
46  /**Serializable オブジェクトの読み込み*/
47      public String read() {
48          try {
49              ObjectInputStream oin = new ObjectInputStream(new
                    BufferedInputStream(in));
50
51              Object obj = oin.readObject();
52
53              if (obj == null) {
54                  Log.d(getClass().toString(), "obj␣is␣null");
55                  return null;
56              }
57              /**受け取ったオブジェクトを元の方に戻す*/
58              Package packageData = (Package)obj;
59              /**各オブジェクトの出力処理*/
60              …
61  /**Serializable からファイルへの書き出し*/
62              FileOutputStream outStream = new FileOutputStream(file);
63              int index = 0;
64              for (byte data : obj) {
65                  outStream.write(data);
66              }
67              outStream.flush();
68              outStream.close();
69              …
```

ソースコード 5.14　読み書きのためのスレッドの一部

5.2.2　DataManager

DataManger では、送信側ではデータの収集、受信側ではデータの配置を行う。送信側では、DataQueue クラスに収集したデータを詰めていき、最終的に BluetoothManager が DataQueue からデータを取り出して送信する仕組みになっている。受信側では、BluetoothManager で通信し受け取ったファイルを各所に配置する。

送信するデータは、文字列データとバイナリデータの 2 種類に分けられる。また、データ取得方法と配置方法にも、Android フレームワークの枠組みで解決するものとそうでないものが存在する。

今回の手法では、扱うデータの種類毎にそれぞれ適切なコードを書かなければならない。しかし、Android の扱っているデータの種類は多岐にわたるため、紙面・実装の都合から上記の条件を網羅したパターンの実装のみ紹介する。

Bookmark の移行

ブラウザブックマークのデータは、ContentProvider を利用することで読み書きが可能である。ブラウザブックマークの読み出し方法をソースコード 5.15 に示す。

```
1   /**ContetProvider を利用して Bookmark データを取得し、
2                   取得したデータをMap に詰めて返すメソッド*/
3   public Map<String,String> collectBookMarks(){
4       Map<String,String> bookMarks = new HashMap<String, String>();
5       /**ContentProvider を利用して Cursor データを取得*/
6       Cursor browser = mContext.getContentResolver().query(Browser.
            BOOKMARKS_URI, strBookmarkProjection, Browser.
            BookmarkColumns.BOOKMARK + "=1", null, null);
7       final int urlIndex = browser.getColumnIndex(Browser.
            BookmarkColumns.URL);
8       final int titleIndex = browser.getColumnIndex(Browser.
            BookmarkColumns.TITLE);
9       /**Cursor の位置を先頭に移動*/
10      if(!browser.moveToFirst()){
11          /**中身が空ならば null を返す*/
12          return null;
13      }
14      /**Cursor の要素の数だけ繰り返しを行う*/
15      do {
16          /**タイトルと URL を取得して、
17          タイトルをkey、URL を value として Map に格納する*/
18          String title = browser.getString(titleIndex);
19          String url = browser.getString(urlIndex);
20          bookMarks.put(title,url);
21      }while (browser.moveToNext());
22      browser.close();
23      /**データを詰め込んだ Map を返す*/
24      return bookMarks;
25  }
```

ソースコード 5.15　BookMarks.java Bookmark データ取得

同様に ContentProvider を利用して、ブラウザブックマークのデータを書き込む方法をソースコード 5.16 に示す。本アプリの実装では、文字列データは全て JSON 形式に変換して送信している。そのため、ソースコード 5.16 では、受信した文字列を JSONObject として解析してから、ContentProvider を利用して書き込みを行っている。

```
1           /**Bookmark データの入った JSONArray を取り出す*/
2           JSONArray elementArray = jsonObject.getJSONArray(
                DataCollector.JSONTAG_BOOKMARKS);
3           for (int i = 0; i < elementArray.length(); i++) {
4               /**Bookmark を Array から１つ取り出す*/
```

```
5                   JSONObject json = elementArray.getJSONObject(i);
6                   /**ContentProvider 用にデータを ContentValues に詰め込む*/
7                   ContentValues values = new ContentValues();
8                   values.put(Browser.BookmarkColumns.BOOKMARK, "1");
9                   values.put(Browser.BookmarkColumns.TITLE, json.getString
                        ("title"));
10                  values.put(Browser.BookmarkColumns.URL, json.getString("
                        url"));
11                  /**データを書き込む*/
12                  context.getContentResolver().insert(Browser.
                        BOOKMARKS_URI, values);
13              }
```

ソースコード 5.16　DataReplacer.java Bookmark データの書き込み

アプリの移行

PackageManager を利用して、端末にインストールされているアプリの一覧を取得する方法をソースコード 5.17 に示す。端末にインストールされている全てのアプリのパッケージ情報を取得するには getInstalledPackages メソッドを利用する。取得した情報を全て取り出すために繰り返し処理を行うが、取得したパッケージ情報には、System アプリと本研究実装の AndroidSudoManager と EnvironmentMigrator も含まれるので、扱っているオブジェクトがこれらであった場合には無視するように実装している。

```
1  String packageName;
2  /**PackageManager のインスタンスを取得*/
3  PackageManager pm = getPackageManager();
4  /**端末にインストールされているアプリパッケージの情報を取得*/
5  final List<PackageInfo> appInfoList = pm.getInstalledPackages(
       PackageManager.GET_UNINSTALLED_PACKAGES);
6  for(PackageInfo info : appInfoList){
7      if(info.applicationInfo!=null){
8          /**EnvironmentMigrator と AndroidSudoManager は除外する*/
9          if(info.packageName.equals(EnvironmentMigrator) || info.
               packageName.equals(AndroidSudoManager))continue;
10         /**System アプリは除外する*/
11         if((info.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM
               )<=0){
12             packageName=info.packageName;
13         }
14     }
15 }
```

ソースコード 5.17　パッケージ名一覧の取得方法

アプリのパッケージを取得する方法を示す。Android アプリのファイル (.apk ファイル) は、/data/app/PackageName.apk として保存されている。apk ファイルは An-

droidSystem 側でバージョン管理されており、実際には/data/app/PackageName-1.apk もしくは、/data/app/PackageName-2.apk という名前で保存されている。バージョン番号である"-1"や"-2"がファイル名に付いていると使い勝手が悪いので、本研究実装では以下のコマンドを実行して EnvironmentMigrator のデータフォルダ (/data/data/EnvironmentMigrator/) 以下に一度バージョン番号をなくしたファイル名でコピーしている。

```
cp /data/app/packagename*.apk /data/data/packagename.apk
```

実際には AndroidSudoManager へ上記のコマンドを送信して実行させている。AndroidSudoManager へのコマンド送信は、ソースコード 5.18 に示した CommandRequestor クラスを利用している。実際のコマンド送信は sendRequest メソッドが担っている。sendRequest では引数で受け取った実行コマンドの文字列を Intent に載せて AndroidSudoManager に送信している。

```
1  public class CommandRequestor {
2      private static final String INTENT_SUDO = "jp.ac.uec.is.spa.hiromu
           .SUDO";
3      private static final String PACKAGE_PATH_PREFIX = "/data/app/";
4      private static final String MIGRATOR_LOCAL_PATH = "/data/data/com.
           example.hiromu.environmentmigrator";
5
6      public static void sendRequest(Context context, String command){
7          Intent intent = new Intent();
8          intent.setAction(INTENT_SUDO);
9          intent.setType("text/plain");
10         intent.putExtra(Intent.EXTRA_TEXT, command);
11         context.startActivity(intent);
12     }
13
14     public static void installApp(Context context, String path){
15         sendRequest(context,"pm␣install␣-f"+path);
16     }
17
18     public static void fileCopyCommand(Context context, String src,
           String dest){
19         sendRequest(context, "cp␣" + src +"␣"+ dest);
20     }
21
22     public static void packageCopy(Context context, String packageName
           ){
23         fileCopyCommand(context, PACKAGE_PATH_PREFIX+packageName+"*apk"
               , MIGRATOR_LOCAL_PATH+"/"+packageName+".apk");
24     }
25
```

```
26      public static void deleteFile(Context context, String fileName){
27          sendRequest(context, "rm␣-f␣"+ MIGRATOR_LOCAL_PATH + fileName);
28      }
29  }
```

ソースコード 5.18　CommandRequestor.java

インストールの実行とファイル配置には CommandRequestor に対して以下のコマンドを適宜発行して実行する。

$pm install -f PackageName //アプリのインストール
$ chmod アクセス権 ファイル名 //アクセス権の設定
$ chown 所有者:所有グループ　ファイル名　//所有者の設定

実行の流れとしては、まずアプリをインストールする。その後データファイルをアプリのデータフォルダに展開し、アクセス権と所有者を設定する。これを繰り返すことによって元の端末のアプリの状態を復元する。

無線 LAN 設定の移行

端末の無線 LAN 設定を移行する手法の実装について議論する。Android の無線 LAN の設定には、WifiManager を利用する。WifiManager の getConfiguredNetworks メソッドで WIFIConfiguration のインスタンスの詰まった List を取得することができる。WIFIConfiguration からは、SSID やネットワーク ID、ネットワークの状態等を取得することができるが、PreSharedKey と WepKey については取得ができず、アクセスしても String で*が返ってくる。

そこで今回は管理者権限を利用して直接設定を書き込んでいるファイルを取得し読み込むことにする。端末に登録されている無線 LAN の設定は `/data/misc/wifi/wpa_supplicant.conf` にソースコード 5.19 の形式で保存されている。

設定毎にこの形式を保っているので、ファイルの先頭から一行ずつ文字列を取り出す。取り出した文字列からさらに `network={` から `}` までの間の文字列を取り出す。そして、=で分割し JSONObject クラスと JSONArray クラスを利用して、ソースコード 5.20 のような JSON 形式に変換して通信に備える。

```
network={
    ssid="SSID"
    psk="preSharedKey"
    key_mgmt=WPA-PSK
}
```

ソースコード 5.19　network の宣言形式の例

```
{network:{
    {ssid:"SSID"},{psk:"preSharedKey"},{key_mgmt=WPA-PSK}
        }
}
```

ソースコード 5.20　network の宣言形式の例

通信で受け取った JSONObject の文字列は 5.16 と同じ方法で解析を行い無線 LAN の設定を行うメソッドに渡す。無線 LAN の設定を行うメソッドをソースコード 5.21 に示す。このメソッドでは、SSID とパスワード、無線 LAN の認証方式を引数に渡すことによって、端末に新たな無線 LAN の設定を追加することができる。

WifiConfiguration クラスのインスタンスを作成し、設定を追加する。しかし認証方式によって設定が異なるため、与えられた認証方式の引数 (auth) に従って設定を分岐させている。

設定を詰めた WifiConfiguration のインスタンスを、WifiManager の addNetwork メソッドに与えて追加する。設定が不十分であった場合等は、ネットワークの追加が失敗して、返り値として −1 が返ってくるので、その場合には Exception を throw するように実装している。

追加が成功した場合には、その設定を saveConfiguration で保存し、updateNetwork でネットワークを更新をする。

```
    private void setNetwork(String ssid, String passwd, int auth) {
        WifiManager wifiManager = (WifiManager) getSystemService(
            Context.WIFI_SERVICE);
        WifiConfiguration addNetConfig = new WifiConfiguration();

        addNetConfig.SSID = "\"" + ssid + "\"";
        switch (auth) {
            case AUTH_NON_SEC :
                【セキュリティ設定の無い無線LAN の設定】
                break;
            case AUTH_WEP :
                【WEP の無線 LAN の設定】
                break;
            case AUTH_WPA_OR_WPA2_PSK:
                addNetConfig.allowedProtocols.set(WifiConfiguration.
                    Protocol.RSN);
```

```
15                addNetConfig.allowedProtocols.set(WifiConfiguration.
                      Protocol.WPA);
16                addNetConfig.allowedKeyManagement.set(WifiConfiguration.
                      KeyMgmt.WPA_PSK);
17                addNetConfig.allowedPairwiseCiphers.set(
                      WifiConfiguration.PairwiseCipher.CCMP);
18                addNetConfig.allowedPairwiseCiphers.set(
                      WifiConfiguration.PairwiseCipher.TKIP);
19                addNetConfig.allowedGroupCiphers.set(WifiConfiguration.
                      GroupCipher.WEP40);
20                addNetConfig.allowedGroupCiphers.set(WifiConfiguration.
                      GroupCipher.WEP104);
21                addNetConfig.allowedGroupCiphers.set(WifiConfiguration.
                      GroupCipher.CCMP);
22                addNetConfig.allowedGroupCiphers.set(WifiConfiguration.
                      GroupCipher.TKIP);
23                addNetConfig.preSharedKey = "\""␣+␣passwd␣+␣"\"";
24                break;
25            }
26            int wifiid = wifiManager.addNetwork(addNetConfig);
27
28            if(wifiid==-1) throw new IllegalArgumentException("Wi-Fi␣
                  setting␣is␣Illegal");
29
30            wifiManager.saveConfiguration();
31            wifiManager.updateNetwork(addNetConfig);
32
33            Log.d(getClass().toString(), String.format("networkid:%d",
                  wifiid));
34        }
```

ソースコード 5.21　無線 LAN の設定方法 (WPA/WPA2-PSK の場合)

第 6 章

評価

6.1 評価実験

提案ツールを利用して評価実験を行った。評価方法は、ブラウザブックマーク、アプリ、無線 LAN 設定を実際に移行できるかどうかによって評価する。アプリの移行については、Google Play のアプリストア [5] から、多くのデータを端末内部に保存するゲームアプリやライフログアプリ、ログインアカウントを持っている SNS のアプリ等合計 10 のアプリを利用して、正常にアプリを移行できるか実験を行った。また、root 化端末対象のアプリである TitaniumBackup、開発者用ツールである ADB のバックアップ機能、パーソナルコンピュータと共に利用するアプリである Helium、Google が提供する環境移行機能である Tap&Go でも同様の環境をバックアップし、移行先端末でリストアすることによってユーザ環境移行を実現できるか実験し比較した。

実験を行った環境を表 6.1 に示す。移行元の端末である Nexus7 から、移行先である Nexus5 へとデータを移行している。利用した端末の OS はそれぞれ、Nexus7 が 4.3(Jelly Bean)、Nexus5 が 4.4(Kitkat) である。ただし、Tap&Go は 5.0(Lollipop) の機能であるので、Tap&Go の実験は Nexus5 を 5.0 へアップデートして実験した。その結果を表 6.2 に示す。

今回の実験対象としたデータセットの場合では、TitaniumBackup はブラウザブックマーク、無線 LAN 設定、アプリの移行全てに成功するという非常に良い結果を示した。ADB のバックアップ機能では、ブラウザブックマークとアプリについては全て移行に成功している。しかし、無線 LAN の設定のみがうまく入らず失敗する結果となった。

Helium の、アプリ E・アプリ J を失敗としているのは、図 6.1 のようにバックアップ非許可となっておりバックアップが実行できなかったためである。実際に非許可となっている部分をタップすると図 6.2 のようなダイアログが提示され、バックアップ対象の開発元からバックアップを機能として提供することが許可されていないためであることがわかる。また明言されていないが、バックアップ・リストア時に現れる画面が ADB Backup のものであることから Helium は、ADB Backup をバックアップの技術として利用していると推察される。しかし、表 6.2 では ADB Backup はアプリ E・アプリ J のバック

表 6.1　実験環境

端末の役割	移行元	移行先
利用した端末	Google Nexus7	Google Nexus5
OS	4.3(JellyBean)	4.4(Kitkat) 5.0(Lollipop)(Tap&Go)
CPU	Qualcomm Snapdragon S4 Pro 1.5 GHz	Qualcomm Snapdragon 800 2.26 GHz
RAM	2GB	2GB

表 6.2　アプリ移行実験結果

利用ツール	TitaniumBackup	ADB	Helium	Tap&Go	提案ツール
アプリ A	○	○	○	×	○
アプリ B	○	○	○	×	○
アプリ C	○	○	○	×	○
アプリ D	○	○	○	×	○
アプリ E	○	○	×	×	×
アプリ F	○	○	○	×	○
アプリ G	○	○	○	×	○
アプリ H	○	○	○	×	×
アプリ I	○	○	○	×	○
アプリ J	○	○	×	×	○
無線設定	○	×	○	○	○
ブックマーク	○	○	○	○	○

アップに成功している。このことから、技術的には Helium も全てのバックアップをすることが可能であると考えられる。

Tap&Go は、ブラウザブックマークと無線 LAN 設定に関しては移行していたが、アプリは全て移行に失敗する結果となった。これについては、アプリ自体は通信を行いダウンロードしてきているがアプリのデータまでは移行出来ておらず、全てのアプリが初期画面からの始まる結果となったため全て失敗とした。

提案ツールについては 6.2 節で議論する。

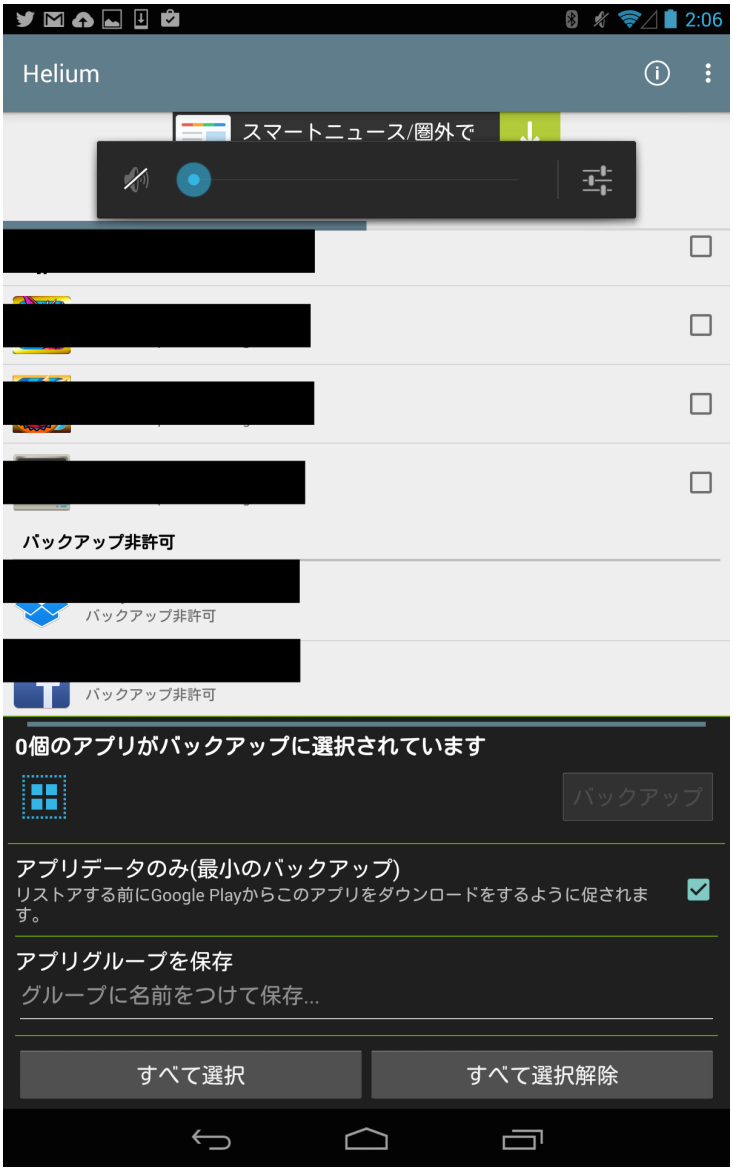

図 6.1　Helium バックアップ選択の画面

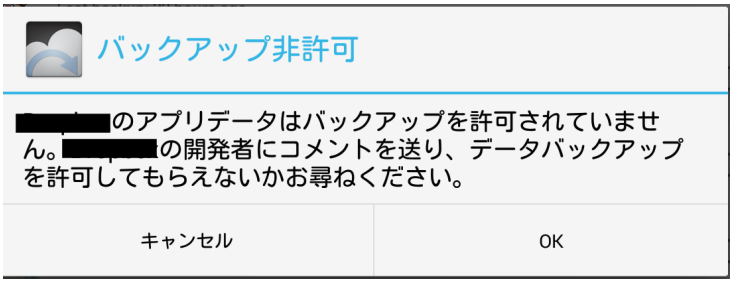

図 6.2　Helium 開発元へのリクエスト画面

6.2 考察

6.1 において、提案ツールが移行に失敗したアプリ E とアプリ H について原因と対策を述べる。まず、移行に失敗したアプリはそれぞれ SNS 系のアプリであったため、移行後にアプリを起動したらログインを求められた。`/data/data/PacakageName` のサイズやファイルの種類を比較したが、元の端末と違いはなかった。つまり、データのコピーはできているが元の端末とそれ以外の点で異なっていたということである。

表 6.2 に示した実験結果からは、Helium ではアプリ E がバックアップに失敗しているが、アプリ H については成功している。さらに、TitaniumBackup や ADB が成功しているところからも、本研究で実装されなかった部分でデータを扱っていると考えられる。

Android では、AccountManager という Android 端末内のアプリで利用するアカウントとパスワードを一括で管理する機能が存在する。端末に保存されているアカウントについては、設定アプリのアカウントの項目から確認できる。

今回移行に成功したツールについてはこの項目に必要なアカウントが追加されていた。しかし、本研究の実装や Tap&Go にはこの項目にアカウントが追加されておらず、Helium についてもアプリ E のアカウントが追加がされていなかった。

AccountManager の設定を移行することによって、これらのアプリについても提案手法での移行が可能になるのではないかと考えられる。また、AccountManager の設定の移行は、パスワードの取得することが API からは出来ない。そのため、無線 LAN 設定の場合と同じように設定ファイルを見つけ、解析し、AccountManager クラスのメソッドを利用して設定を反映する。無線 LAN の設定が可能であったことからも、提案手法でこれを実現することが可能なことは明らかである。

第 7 章

おわりに

本研究では、Android 端末におけるユーザ環境の移行を実現するために、管理者権限機構の改良手法とそれを利用したデータ移行手法の提案を行った。また、それを実現する AndroidSudoManager と EnvironmentMigrator の 2 つの Android アプリの実装を行った。残念ながら、Android の機能全てを網羅して完全なデータ移行を遂行するツールの完成には至らなかった。しかし、評価実験より本提案システムを利用する事によって、完全なデータ移行を行うことは十分に可能であることを示した。

今後の展望としては、EnvironmentMigrator の完成が挙げられる。今回実装しなかった細かな設定や壁紙設定、アカウント設定等の移行機能の実装することで、提案ツールの完成を目指す。本提案ツールが完成することによってユーザ環境の移行が可能になり、端末移行時の利用者の時間的なコストが削減されることが期待される。

謝辞

本研究を遂行するにあたっては、いろいろな方々にお世話になりました。

まず、主任指導教員の多田好克先生には日頃から熱心なご指導、そしてご鞭撻を賜わりました。また、ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり、大変貴重なご助言をいただきました。ここに厚く御礼申し上げます。

指導教員の小宮常康先生には本研究の実装について大変貴重なご助言を頂きました。心より感謝いたします。

そして、本研究が行なえたことは、研究方針や方法論について議論をし、共に研究生活をおくってきた基盤ソフトウェア学講座の学生諸氏おかげでもあります。最後に、これらの皆さんに感謝いたします。

参考文献

[1] Mohammad Nauman, Sohail Khan and Xinwen Zhang : "Apex: extending Android permission model and enforcement with user-defined runtime constraints," *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security ASIACCS '10*, pp.328-332, ISBN: 978-1-60558-936-7

[2] James Sellwood and Jason Crampton : "Sleeping android: the danger of dormant permissions," *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices SPSM '13*, pp.55-66, ISBN: 978-1-4503-2491-5

[3] David Barrera, Jeremy Clark, Daniel McCarney and Paul C. van Oorschot : "Understanding and improving app installation security mechanisms through empirical analysis of android," *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices SPSM '12*, pp.81-92, ISBN: 978-1-4503-1666-8

[4] "Android のアップデート:Nexus と Google Play エディションの端末", `https://support.google.com/nexus/answer/4457705?hl=ja&ref_topic=3415518`, 2015 年 1 月 13 日参照

[5] "Google Play", `https://play.google.com/store/apps?hl=ja`, 2015 年 1 月 17 日参照

[6] "Titanium Backup-Titanium Track", `http://www.titaniumtrack.com/titanium-backup.html`, 2015 年 1 月 20 日参照

[7] "clockworkmod-Helium", `https://www.clockworkmod.com/carbon`, 2015 年 1 月 20 日参照

[8] "Android のセキュリティ構造", `http://source.android.com/devices/tech/security/`, 2015 年 1 月 20 日参照

[9] "sudo コマンド", `http://www.sudo.ws/`, 2015 年 1 月 26 日参照