GUI構造に基づいた Androidアプリケーション動的解析支援の検討

塩治 榮太朗 †　　秋山 満昭 †　　岩村 誠 †　　針生 剛男 †

†NTT セキュアプラットフォーム研究所
180-8585 東京都武蔵野市緑町 3-9-11
{shioji.eitaro, akiyama.mitsuaki, iwamura.makoto, hariu.takeo}@lab.ntt.co.jp

あらまし Android アプリケーションの多くは GUI を前提としているため、動的解析時にはタッチ操作などのユーザイベントを発生させて実行を促す必要がある。しかし、マルウェア検査時などの不特定多数のアプリケーションを自動解析する状況においては対象の仕様が明らかではないため適切なイベントを事前に用意することが難しく、また、無作為なイベント発生は非効率的である。そこで本研究では、OS 内部で管理されている GUI 構造データの解析から画面遷移モデルを構築し、それに基づいた効率的なユーザイベントの動的生成手法を提案する。これにより、動的解析時のコードカバレッジの向上を目指す。

Assisting Dynamic Analysis of Android Applications based on GUI Structure

Eitaro Shioji†　　Mitsuaki Akiyama†　　Makoto Iwamura†　　Takeo Hariu†

†NTT Secure Platform Laboratories
3-9-11 Midori-Cho, Musashino-Shi, Tokyo 180-8585 JAPAN
{shioji.eitaro, akiyama.mitsuaki, iwamura.makoto, hariu.takeo}@lab.ntt.co.jp

Abstract Android applications commonly assume GUI, requiring user-generated events such as taps for stimulating execution during dynamic analysis. However, under circumstances of analyzing unknown applications, e.g., inspecting for malware, it is difficult to prepare in advance the exact set of required events because their specifications are not known, and random generation is inefficient. Thus, in this paper we discuss a method that analyzes the GUI structure information stored in the OS to construct a transition model, which efficiently guides the dynamic event generation. Our aim is to improve code coverage of dynamic analysis.

1　はじめに

近年の Android 端末の普及に伴ってアプリケーション（以降アプリ）の数も急増しており、公式マーケットのみに注目しても 50 万を超える種類 [1] が配信されている。しかしその一方で、悪性な挙動を行うアプリ（以降マルウェア）による個人情報漏洩などの被害が懸念されており、実際に公式マーケット上においてもマルウェアが発見されている [4]。このようなアプリを検知・排除するためにはその解析を行う必要があるが、数が膨大であるために手動での検査には限界があり、ある程度自動化させた手法が必要となる。

よって、実際にアプリを実行させて挙動を解析する動的解析が有効であると考えられる。特にAndroidではGUI(Graphical User Interface)を前提としたアプリが多いため、実行時にはタップ（画面を指で叩いて短時間で離す）操作やテキスト入力などのユーザイベントを模擬して入力する必要がある。しかし、ソフトウェア開発におけるテスト工程などとは違い、不特定多数のアプリを自動解析する状況においては対象の仕様が明らかではないため、適切なイベントを事前に用意することが難しい。また、無作為なイベント発生は効率的ではない。

そこで本研究では、プログラムのコード実行率（以降、コードカバレッジ）の効率的な向上を目的とし、OS内部で管理されているGUI構造データの実行時の解析から画面遷移モデルを構築しつつ、それに基づいてユーザイベントを効率的に動的生成する手法についての検討を行う。さらに、本システムのプロトタイプを用いた実行例も示し、課題や今後の取り組みについて述べる。また、本取り組みのもう１つの目的として、本解析によって副次的に得られる状態遷移に関する情報を別の解析において再利用することについても検討する。

本文では以降、２章で背景、３章で提案手法、４章で実装・実行例、５章で議論、６章でまとめについてそれぞれ述べる。

2　背景

本節では、本研究の範囲、関連研究、およびAndroid上のGUI情報について述べる。

2.1　研究の範囲

本研究はプログラムの動的解析時にその実行を促しより多くの挙動を引き出すための入力を与えることを目的とする。これはPC上のプログラムの動的解析時においては従来からの課題でもある[12][13]。このような入力の種類としては例えば外部サーバとのネットワーク通信や、OSのバージョンや時刻などの環境情報などが挙げられ、またAndroidではその特有の仕組みであるIntent等も考慮する必要がある。本研究では、Androidで特に重要となる画面上のユーザインタラクションによる入力に焦点を置く。なお、公式マーケットではBouncer[2]と呼ばれる自動検査システムの導入が発表されているため、今後は起動のみで悪性な挙動を行うマルウェアは減少すると考えられ、このような実行促進を行うことの必要性が増すと考えられる。

2.2　関連研究

Androidアプリのセキュリティ検査に関する研究においては、Monkey[5]という、ユーザインタラクションを含む様々なイベントを無作為に発生させるツールがよく利用されている[14][10]。また、本取り組みはアプリの自動テストと関連が深いが、テストではその仕様が事前に明らかであるため、モデルやテストケースの作成が部分的に手動で行われるものが多い[8]。文献[9]では本研究と似たアプローチを取っているが、状態の決定方法などが異なっている。

2.3　Android上のGUI情報

AndroidのGUIはViewクラスを継承するオブジェクトをノードとして持つ木構造のデータ(以降、View Hierarchy)[3]としてOS内部で管理されている。図1にて、あるGUI画面とそれに対応するView Hierarchyのイメージを例として示す。木構造において、節ノードはレイアウトに関するコンポーネント(FrameLayout、LinearLayoutなど)を、葉ノードはユーザとのインタラクションが発生するコンポーネント(TextView、EditText、Buttonなど)を表す。なお、View Hierarchyの指定はAPKファイル内のXMLファイルに記述しプログラムに読み込ませることもできるが、プログラム内から動的に操作することができたり、実行時にのみ決定される情報もあるため、静的解析によって事前に全て把握できるわけではない。なお、Android SDKに同梱されているツールであるHierarchy Viewer[17]を利用すると、現在のデバイス画面上のView Hierarchyを確認することができる。

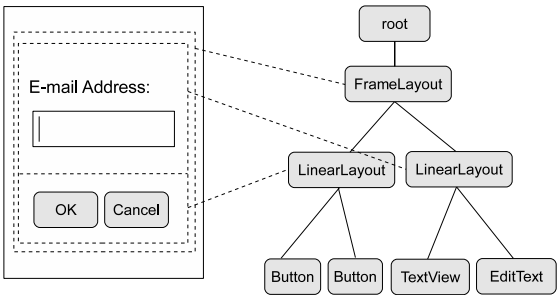

図 1: View Hierarchy の例

3　提案手法

本節では、提案するユーザイベントの動的生成手法について述べる。図 2 のように、画面の状態をノード、ユーザイベントをエッジとしたグラフを構築し、それを巡回する。以下、このグラフにおける状態の決定方法と巡回アルゴリズムについて説明する。

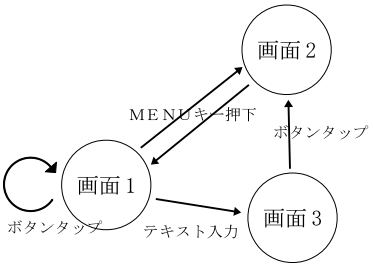

図 2: 画面状態遷移グラフ

3.1　状態の決定

3.1.1　画面状態

状態を定義するにあたり、実行中のプログラムに関する情報の中で何を状態の決定に含めるかを決めることが重要となる。本手法では、画面遷移に関わる状態を扱うため、それに特化した状態の決定方法を用いる。ここで含む情報が過剰であると不要な状態が増加してしまう一方、不足していると必要な状態やイベントの区別ができなくなるため、適切な決定方法が必要である。

例えば、図 3 で示される Android4.0.3 標準の電卓アプリの例において、画面 (a) から (c) への変化を異なる状態への遷移として認識することは妥当である。一方で (a) から (b) での、数字ボタンを押した際に変化した結果表示部の内容を状態決定に用いると無数の状態が生成されてしまう。

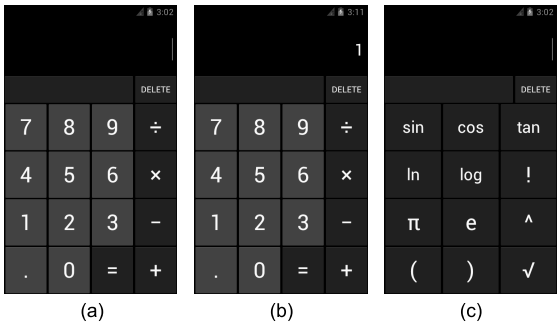

図 3: 状態識別の例

3.1.2　状態の決定方法

まず、得られた View Hierarchy におけるノードの中から、タップの対象となるノードを決定する。これは主に、View Hierarchy における各 View の属性情報や木構造における位置によって決定する。具体的には、画面外に配置されていないか、非表示指定になっていないか、タップに反応するか、などの属性情報に関する条件を満たし、かつ、木構造において葉ノードとなっているものを選択する。このようにして選択された各対象ノードとルートノードまでの経路に存在しないノードとエッジを取り除いて得られる木を最終的に状態決定に利用する構造とする。この情報に、パッケージ・アクティビティ名やテキストボックスへの入力が埋まっているかなどの情報をさらに付加して状態を決定する。例として、ある画面から実際に得られた View Hierarchy のダンプの一部を示す。

```
0: com.android.internal.policy.impl.Pho...
    1: android.widget.FrameLayout
      2: android.view.ViewStub
      2: android.widget.FrameLayout
        3: android.widget.LinearLayout
          4: android.widget.LinearLayout
            5: android.widget.LinearLayout
```

```
      6: [*] android.widget.Button
      6: android.widget.Button
      6: [*] android.widget.Button
  4: android.widget.FrameLayout
    5: android.widget.FrameLayout
  4: android.widget.LinearLayout
    5: android.widget.ScrollView
      6: android.widget.TextView
  4: android.widget.LinearLayout
    5: android.view.View
    5: android.widget.LinearLayout
      6: com.android.internal.w...
      6: android.widget.ImageView
    5:android.view.View
```

詳しい属性情報は省略しているが、タップの対象として判定されたものが記号 [*] が付加されているノードである。また、各行の左の数字は木における深さを表しており、各行の文字列はクラス名を表している。次に、ここから選定された各ノードとルートノードまでの経路間に存在するノード以外を消去した結果を示す。

```
0: com.android.internal.policy.impl.Pho...
    1: android.widget.FrameLayout
      2: android.widget.FrameLayout
        3: android.widget.LinearLayout
          4: android.widget.LinearLayout
            5: android.widget.LinearLayout
              6: [*] android.widget.Button
              6: [*] android.widget.Button
```

この情報から木構造に関する情報のみを抽出して文字列化したデータ “0,1,2,3,4,5,6*,6*” に、パッケージ名、アクティビティ名、テキストボックスの入力状態（全て埋まっているか否かの 2 値）等を連結したのちハッシュ化したものを状態として識別する。なお、ノードを間引く理由はノイズカットであり、実質的には等価であるような状態が重複して登録されることを防いでいる。

3.2 グラフの巡回

3.2.1 巡回アルゴリズム

実行の各時点において得られている画面状態遷移グラフに対する深さ優先探索に基づいた巡回を行っている。以下に巡回アルゴリズムの概要を示す。

Step 1: 現在の状態を取得し、それが未知の状態である場合、グラフにノードを追加する。また、前回実行したイベントが存在する場合、前回の状態から現在の状態に対するエッジを追加する。

Step 2: 現状態に未実行のイベントが存在する場合、そのイベントを実行したのち Step 1 に戻る。

Step 3: 次に実行するイベントを、現状態のイベントへのスコアリングに基づいて決定する。スコアリングは各イベントの遷移先の状態から深さ i までの探索において発見された未実行イベントの総数であり、$i = 1$ を初期値とする。イベントの中で最大のスコアを持つもののスコアが非ゼロである場合、そのイベントを実行して Step 1 に戻る。

Step 4: ここで、現状態から到達可能なノードまでのホップ数の最大値である D_{max} を計算し、$i < D_{max}$ である場合、i をインクリメントして Step 3 に戻る。

Step 5: 現状態が開始時の状態と同一である場合は解析を終了する。そうでない場合、BACK ボタンを押下し Step 1 に戻る。

なお、標準的な Android アプリにおいては BACK ボタンはスタック構造として積まれた Activity の遷移履歴を巻き戻す操作として使われているため[1]イベントとしては扱っていない。また、Step 3 のスコアリングにおける、各イベントの遷移先の状態からの探索においては現状態を経由することにより各イベントのスコアが類似してしまうのを防ぐため、現状態へのエッジは実行対象外としている。なお、本アルゴリズムが実行イベント数よりも現状態からの近さを優先するのは、遷移に時間的コストが発生することを前提としているためである。

3.2.2 例

図 4 にて、イベント選択の例を示す。実線の矢印は実行済みイベントの遷移先、点線は未実行イベントを表し、灰色の円は現在の状態のノー

[1]独自のイベントハンドラを登録することもできるため、必ずしもその限りではない。

ド、黒い円、白い円はそれぞれスコア計算に考慮される状態のノードとそうでないものを表し、括弧内の数字はそのイベントのスコアを表す。状況 (a) では現状態のノードから深さ 1 のノード v_2 のみに未実行のイベントが存在するため、v_2 へ遷移するイベント e_2 が選択される。一方、状況 (b) では深さ 1 のノードのイベントは全て実行済みであるため、深さ 2 のノードまで探索が行われた結果、未実行のイベントの数が最も多いイベント e_3 が選択される。なお、この例は途中で停止した解析のグラフデータを継続しつつ解析を再実行するような状況を想定している。

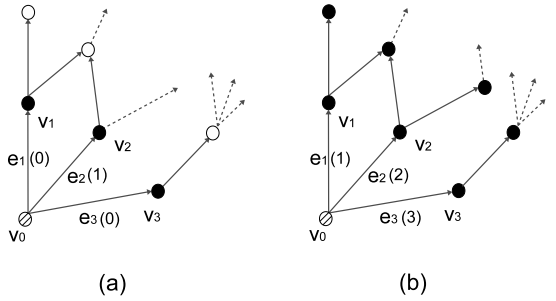

図 4: イベント選択例

4 実装・実行例

上記で提案した手法に基づいた動的解析システムのプロトタイプを実装した。本節では実装についての説明と、その実行例および簡易的な評価の結果について述べる。

4.1 実装

まず、本システムの概要を図 5 に示す。大まかな処理の流れとしては図に示される数字の順で、以下ようになっている。

1. 現在の GUI 構造に関する情報の取得を GUI 構造情報を扱うモジュールに適切なタイミングで依頼する。
2. WindowManager Service から View Hierarchy のダンプを取得する。これは Hierarchy Viewer[17] が内部的に行っている仕組みと同じである。
3. この情報を解析した後、遷移グラフデータベースを必要に応じて更新する。
4. 蓄積されたグラフデータを解析し次に実行させるイベントを決定する。
5. イベントの発生を monkeyrunner[6][2] ライブラリ経由で行い、また、必要に応じて実行環境の情報やスクリーンショットの取得などを行う。その後、必要に応じてグラフデータを更新し、1. へ戻る。

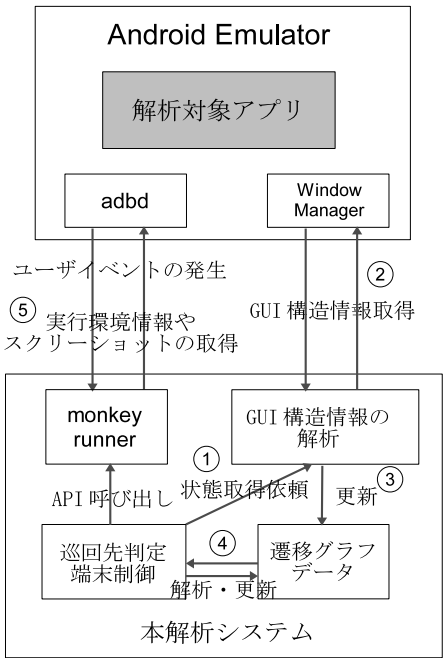

図 5: システム概要図

現時点で対応しているイベントはタップ、メニューボタンの押下、およびテキストボックスへの入力である。テキストボックスに関しては、数字のみが入力可能になっていたり、メールアドレスのみを受容するケースが多いため、本実装では常に文字列 “1234 @ abcd.com” を入力するようにしている。また、Intent の利用により別パッケージの Activity を呼び出すアプリが多く存在するため、解析範囲を対象アプリ内に留めるために別パッケージに遷移した場合は無条件で BACK ボタンを押下することにより元のパッケージへの復帰を試みている。

[2] Monkey とは異なるツールである。

4.2 実行例

参考までに、オープンソースのメールクライアントアプリである K-9 Mail[15] を用いた本システムの実行例を示す。実行環境は、Intel Core i7 2.13GHz、 4GB RAM、Windows 7 SP1、Android Emulator 上の Android 4.0.3 であり、ネットワークには接続していない。

15 分間の実行において得られたコードカバレッジについて、本手法とランダムな生成方法である Monkey を比較した結果が表 1 に示されている。列は左から実行されたクラス数、メソッド数、基本ブロック数、行数の割合をそれぞれ表している (Monkey は 3 回平均)。また、コードカバレッジの計測には EMMA[16] を利用した。生成されるイベントの質に関わるオプションとしては、遷移できるパッケージを限定させる “-p com.fsck.k9” を付加して実行している。

提案手法での実行イベント数は 41 であり、その過程で得られた状態数は 22 であり、一方 Monkey は約 2400 回のイベントが発生していた。結果として Monkey よりも優れたカバレッジが、より少ないイベント数で得られた。なお、速度に関する最適化はほとんど行っておらず、Monkey では Intent などの GUI 以外のイベントも含まれている。また、現時点での実装課題とコードカバレッジ取得方法の都合により具体的な計測値は得られていないが、より長時間の実行ではより大きな差が得られることが分かっている。

表 1: コードカバレッジの比較 (15 分)

	class	method	block	line
提案手法	27%	22%	14%	15%
Monkey	11%	8%	5%	6%

5 議論

本節では、本システムの現段階での制限と今後の課題、二次的な解析への利用、および関連研究について述べる。

5.1 制限・今後の課題

提案手法の評価・改良 状態決定方法、巡回アルゴリズムともに、より適切な評価を行う必要がある。評価対象のアプリについても増やす必要があるが、現在はコードカバレッジの測定に利用している EMMA がソースコードを必要とするため、APK ファイル単体の場合は測定することが難しい。よって、ソースコードに依存せずにバイトコードの粒度でカバレッジを計測するような手法と組み合わせる方法などが必要である。さらに、状態の決定方法についても、その妥当性について評価する必要がある。

また、巡回アルゴリズムにおけるスコアリングの結果、少数の状態間を無限ループするような事象が発生することがあるため、ループ検知や、遷移回数に応じたスコアの重み付けを行うなどの改良やが求められる。対応していないフリックや長押し操作などの他のユーザイベントへの対応も必要である。

実行速度 エミュレータ上で実行することによる実行速度の低下に加え、イベントを発生させた後に遷移が終了し画面が安定するまで数秒間のスリープを挟んでいるためにさらに速度が下がる。また、monkeyrunner が利用している adb が不安定であり定期的に応答しなくなるなどの問題があるため、これを検知して再起動させるなど、ロバスト性を高める必要がある。

この問題の解決および、更なる効率化のため、巡回アルゴリズムにおける現状態内のイベント選択 (Step 2) での順序をランダム行うように変更した上で、複数の実行環境で並列に一定時間実行させた結果をマージじ、さらにそのマージ結果からの再開実行を並列に行うことを繰り返すことにより加速させる手法を検討している。また、Android の WindowManager による GUI 構造のダンプが低速であるため、この部分を高速化して差し替えるアプローチも有効であると考えられる。

特殊な View View の中には、例えば内部で OpenGL を利用するものや、Web ブラウザとして機能するものなどが存在するが、本手法で

はあくまでも View Hierarchy の構造に基づいた解析を行っているため、それらの内部における GUI コンポーネントの識別・実行には対応できない。対応するためには、それらの内部まで踏み込んだ解析を行う必要がある。

状態　本手法では、あくまでも GUI 構造に絞った状態の決定を行っているため、それ以外の要素の変化によってプログラムの画面遷移が変化する場合に既に得られているグラフとのずれが生じる。例えば同じ状態から同じイベントを実行しているにもかかわらず、それまでの遷移履歴に応じて異なる状態に遷移する場合が考えられる。このような状態のずれを検知し、自動的にグラフを補正していくような機能についても検討したい。

ホワイトボックス解析　本研究では、実行時に得られる情報のみに基づいたブラックボックス解析を行ったが、バイトコードやマニフェストファイルの内容を利用することにより向上が可能であると考えている。例えば、詳細は明らかにされていないが文献 [11] のように、バイトコードやその逆コンパイル結果からメソッドのコールグラフを生成し利用することが考えられる。ただし、難読化が行われている場合には上手く行うことができない可能性もある。

なお、ホワイトボックスの結果を利用して強制的にコードを実行させるという方法との差異として、本手法ではユーザによる操作によって実行されうることを保証できるという点が挙げられる。

5.2　二次解析への利用

本システムによる解析で得られた画面状態遷移グラフデータは別の動的解析を行う際に役立つことが想像できるが、静的解析においても有用となる利用方法があると考えている。Android アプリには悪性かどうかの自動判断が困難であるような挙動を示すものも多く、そのようなアプリについては最終的には人による手動の解析・判断が必要となる。そこで、本解析で得られた結果を用いることでそのような解析の補助情報の生成に利用できる。これは、例えば logcat や TaintDroid[18][3]のログを本手法で得られた画面状態遷移グラフを可視化したものと重ね合わせることにより、例えば確認ダイアログの有無などを考慮した、前後の文脈を踏まえた上での挙動の悪性判定を行うことができる。図 6 に、実際に本システムによって自動的に得られたグラフとスクリーンショットの一部を用いて構成した図を利用イメージとして示す。

6　まとめ

本稿では GUI 構造に基づいて効率的にユーザイベントを自動生成させる手法についての検討を行い、その状態決定手法と巡回アルゴリズムを提案した。また、実装したプロトタイプを用いた実行例を示した。今後は抽出した課題に基づき本システムの評価・改良を行うとともに、ユーザイベント以外の入力も組み合わせていき、コードカバレッジの更なる向上を目指す。

参考文献

[1] Mobile Statistics, http://www.mobilestatistics.com/mobile-statistics/

[2] Google Mobile Blog, “Android and Security,” http://googlemobile.blogspot.jp/2012/02/android-and-security.html

[3] Android Developers, “View,” http://developer.android.com/reference/android/view/View.html

[4] 日経 BP 社, “個人情報を盗む「the Movie」アプリは 29 種類、数百万人が被害の恐れ,” http://pc.nikkeibp.co.jp/article/news/20120417/1046202/

[5] Android Developers, “UI/Application Excerciser Monkey,” http://developer.android.com/tools/help/monkey.html

[6] Android Developers, “monkeyrunner,” http://developer.android.com/tools/help/monkeyrunner_concepts.html

[3] TaintDroid はテイント解析を用いた詳細なデータフロー追跡を行う。

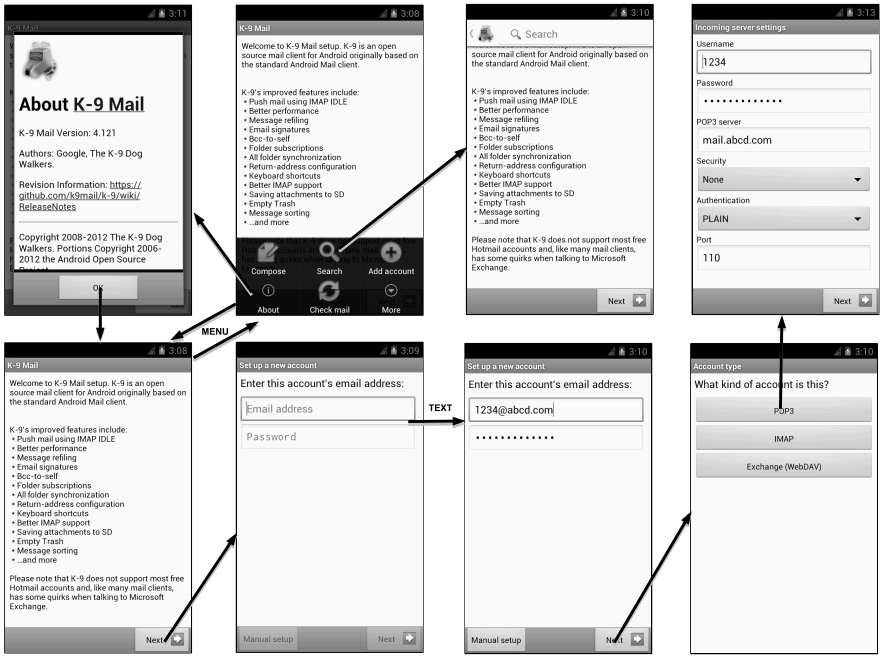

図 6: 画面遷移の可視化

[7] Nicholas J. Percoco and Sean Schulte, "Adventures in BouncerLand," In Proceedings of the 15th Black Hat USA, 2012.

[8] T. Takala, M. Katara, and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," In Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST'11), 2011.

[9] D. Amalfitano, A. Fasolino, and P. Tramontana, "A GUI Crawling-based technique for Android Mobile Application Testing," In IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011.

[10] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," In Proceedings of the 6th International Workshop on Automation of Software Test (AST '11), 2011.

[11] R. Mahmood et al., "A whitebox approach for automated security testing of Android applications on the cloud," Automation of Software Test (AST), 2012.

[12] J. R. Crandall et al., "Temporal search: detecting hidden malware timebombs with virtual machines," In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII), 2006.

[13] D. Brumley et al., "Automatically identifying trigger-based behavior in malware," In Botnet Detection, volume 36 of Countering the Largest Security Threat Series: Advances in Information Security. Springer-Verlag, 2008.

[14] R. Xu, H. Saidi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," In Proceedings of the USENIX Security Symposium, 2009.

[15] K-9 mail, `http://code.google.com/p/k9mail/`

[16] EMMA: a free Java code coverage tool, `http://emma.sourceforge.net/`

[17] Android Developers, "Hierarchy Viewer," `http://developer.android.com/tools/help/hierarchy-viewer.html`

[18] W. Enck et al., "TaintDroid: An information-flow tracking system for real-time privacy monitoring on smartphones," In 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010.