Uma Técnica Prognóstica para Desenvolvimento Seguro de Aplicativo Android

Ricardo Luis Dias Martins Ferreira
Military Institute ofEngineering (IME), Brazil
ricardo.ldm.ferreira@gmail.com

Anderson F. P. dos Santos
Military Institute of Engineering (IME), Brazil
anderson@ime.eb.br

Ricardo Choren
Military Institute of Engineering (IME), Brazil
choren@ime.eb.br

***Resumo*—A procura por vulnerabilidades em aplicativos (app) para Android, através de uma abordagem baseada no dex bytecode do app, tem sido objeto de estudo de diversos trabalhos. Esta abordagem, denominada deteção tardia, que ocorre quando a aplicação está pronta, geralmente não identifica uma vulnerabilidade antes que diversos usuários já estejam expostos. Este trabalho apresenta uma técnica, baseada em análise estática com casamento de padrões, para identificar vulnerabilidades de forma antecipada, durante o processo de codificação do aplicativo. Esta técnica foi avaliada através de uma prova de conceito experimental com aplicativos open-source, analisados pela ferramenta appDroidAnalyzer, que identificou diversas aplicações com potenciais vulnerabilidades em seu código-fonte.**

***Palavras Chave*—Analise estática, Android, Segurança, Vulnerabilidade.**

***Abstract*—Searching for vulnerabilities in Android apps through approaches based on the app's dex bytecode has been applied to a lot of researches. This approach, called late detection, is applied to apps already released, and usually doesn't identify vulnerabilities before users have been exposed. This article presents a method based on static analysis with matching patterns for identifying these vulnerabilities beforehand, during the app development, avoiding users' exposure. The presented technique was evaluated by an experimental test proof applied to open-source applications, analyzed by appDroidAnalyzer, identifying dozens of apps affected by vulnerabilities in their source code.**

***Index Terms*—Android, Security, Static analysis, Vulnerability.**

I. INTRODUÇÃO

Dispositivos móveis com o sistema operacional (SO) Android dominam o mercado mundial de vendas de smartphones. Segundo o International Data Corporation (IDC), no terceiro trimestre de 2016, 86,8% dos aparelhos possuíam este SO instalado [1]. Este crescimento é acompanhado pelo número de aplicativos (apps) disponíveis para download na loja oficial Google Play, onde aproximadamente 2,72 milhões de apps estão catalogados [2].

De acordo com o *5th Annual State of Application Security Report, 90%* dos *apps* mais populares de saúde e finanças avaliados possuem pelo menos dois riscos de segurança listados em *OWASP Mobile Top Ten Risks* [3]. Esta lista é mantida pelo projeto OWASP Mobile Security Project, que é focado em verificações de segurança para aplicações móveis e possui como finalidade prover os meios necessários para construir e manter aplicações móveis seguras. O seu principal objetivo é o de minimizar os impactos e a exploração de vulnerabilidades. Considerando o impacto no ativo explorado, a criticidade e a facilidade de exploração e de deteção, esta lista contém a seguinte classificação: *M1. Weak Server Side Controls; M2. Insecure Data Storage; M3. Insufficient Transport Layer Protection; M4. Unintended Data Leakage; M5. Poor Authorization and Authentication; M6. Broken Cryptography; M7. Client Side Injection; M8. Security Decisions Via Untrusted Inputs; M9. Improper Session Handling; e M10. Lack of Binary Protections* [4].

Para ilustrar este cenário, o aplicativo Stick Cricket, um jogo de cricket com mais de 200.00 downloads, é vulnerável quanto ao armazenamento de informações no dispositivo, uma vez que a pontuação obtida em uma partida pode ser alterada diretamente no arquivo de preferências. Esta vulnerabilidade está classificada pela *OWASP em M2 - Insecure Data Storage*. Este exemplo demonstra que não existe uma grande preocupação, de forma preventiva, com a segurança dos aplicativos desenvolvidos.

As vulnerabilidades que ocorrem em um *software*, geralmente, são originadas por falhas ocorridas durante o processo de desenvolvimento [5]. Vulnerabilidades são brechas ou falhas em um sistema que permitem que usuários mal-intencionados violem os aspectos de segurança de uma aplicação [6].

A identificação de vulnerabilidades em aplicativos para a plataforma Android é um tema tratado em diversos trabalhos. Em [7], [8] e [9] a procura por vulnerabilidades ocorre através de engenharia reversa do dex bytecode do aplicativo, após o *app* estar pronto e publicado, o que já pode ter afetado milhares de usuários. Esta abordagem será denominada neste trabalho como deteção tardia de vulnerabilidade, que significa encontrá-la após a criação e compilação de um programa. Segundo [10], somente quando uma versão completa do software está pronta é que o desenvolvedor utiliza uma ferramenta para detetar vulnerabilidades em seu código, e, ao receber o relatório com a identificação dos problemas, estes são corrigidos, gerando uma nova versão do sistema. Em geral, não oferece uma prevenção para que as vulnerabilidades estejam presentes software.

Já em [10] a abordagem utilizada é baseada em encontrar vulnerabilidades durante a etapa de desenvolvimento em aplicações para Web. [10] utiliza a técnica de deteção antecipada que procura vulnerabilidades em aplicações durante o processo de codificação, enquanto o programador está criando ou editando código-fonte. Em [11] procura-se identificar problemas de desempenho relacionados com o gerenciamento da memória

e relacioná-los com a sua baixa reputação na loja Google Play.

Este trabalho propõe uma técnica para descobrir antecipadamente, durante a etapa de desenvolvimento, potenciais vulnerabilidades em códigos de aplicativos para a plataforma Android. Isto propicia que estas sejam descobertas em um momento anterior à publicação para *download*, evitando que os usuários do app sejam afetados. É baseada em uma classificação de vulnerabilidades que, apoiada na lista *OWASP Mobile Top Ten Risks*, identifica os métodos em Java que permitem que a vulnerabilidade seja inserida no código-fonte e indica quais os princípios de segurança definidos pela norma ABNT NBR ISO/IEC 27.002:2013 serão violados.

Para avaliar a técnica apresentada foi desenvolvida a ferramenta appDroidAnalyzer, que foi implementada a partir de uma extensão do Lint e utilizou a técnica de análise estática com casamento de padrões para identificar um código perigoso.

Este trabalho está organizado da seguinte forma: o capítulo I apresentou a introdução, onde está descrito o problema a ser enfrentado e uma breve descrição da solução que será proposta; o capítulo II descreverá a abordagem utilizada, a técnica para avaliação e deteção de código e a classificação utilizada como base; já o capítulo III descreverá a implementação da ferramenta appDroidAnalyzer; o capítulo IV exibirá a análise realizada para avaliar a técnica; no capítulo V serão descritos os trabalhos relacionados; e, finalmente, no capítulo VI serão apresentadas as conclusões.

II. ABORDAGEM

O objetivo deste trabalho é ajudar o desenvolvedor de aplicativos para a plataforma Android a descobrir, antecipadamente, durante o processo de desenvolvimento, as possíveis vulnerabilidades contidas no aplicativo que ele está criando.

Existe um grande número de recomendações em programação que aliam a segurança de uma aplicação com boas práticas de programação, que de fato, se seguidas, eliminam muitos erros e falhas de segurança das aplicações [12]. Dentre estas recomendações, em relação à segurança, existem diversas normas e outras iniciativas que contribuem para melhorar a segurança nas aplicações, como o Código de Prática para Controles de Segurança da Informação (ABNT NBR ISO/IEC 27.002:2013), Common Criteria for Information Technology Security Evaluation (ISO/IEC 15408), Building Security In Maturity Model (BSIMM-7), Software Engineering Institute CERT Coding Standards da universidade Carnegie Mellon e OWASP Mobile Security Project.

Para alinhar estas recomendações com o objetivo deste trabalho foram utilizadas as orientações estabelecidas pela norma ABNT NBR ISO/IEC 27.002:2013 e a lista de vulnerabilidades da organização OWASP.

A norma ABNT NBR ISO/IEC 27.002:2013 (Código de Prática para Controles de Segurança da Informação) [13] foi selecionada porque define os objetivos e diretrizes gerais para manter um sistema de gestão de segurança da informação. Contém recomendações de boas práticas para implementar este sistema definindo os pilares de segurança que serão utilizados: confidencialidade, integridade e disponibilidade.

A lista *OWASP Mobile Top Ten Risks* [3] foi escolhida porque, fundamentada em sua metodologia, classifica e expõe os riscos encontrados com mais frequência dentre os dados coletados por diversas organizações que contribuem para este projeto. Além disto, esta seleção considerou que esta lista é referente a vulnerabilidades específicas para aplicações desenvolvidas para plataformas móveis.

À luz do objetivo proposto foi realizada uma análise a partir da qual puderam ser identificadas as características de cada vulnerabilidade que podem ser evitadas, preventivamente, no decorrer do processo de construção do app e que estivessem diretamente relacionados com a sua ocorrência no dispositivo móvel.

Portanto, M1, M3, M6 e M10, descritas em [4] não farão parte do escopo da classificação de vulnerabilidades. M1 está relacionada com as validações de segurança que devem ser verificadas no lado servidor de aplicações cliente-server. M3 preocupa-se com os problemas de segurança que ocorrem na comunicação de rede entre um aplicativo e o servidor. M6 é referente à criptoanálise que o algoritmo de criptografia utilizado para armazenar ou trafegar informações do aplicativo pode sofrer. M10 está relacionada com as recomendações que devem ser seguidas para evitar a engenharia reversa do app, que tem como objetivo alterá-lo para se obter alguma vantagem.

Desta forma, a classificação sugerida relaciona os riscos para aplicações móveis identificadas e analisadas pela *OWASP Mobile Top* Ten Risks com os métodos da linguagem Java que podem gerar vulnerabilidades associadas com estes riscos. Em seguida os autores correlacionaram esta análise com os princípios de segurança de confidencialidade, integridade e disponibilidade que são definidos pela norma ABNT NBR ISO/IEC 27.002:2013 [13]. A partir desta classificação as vulnerabilidades foram agrupadas em duas categorias: Armazenamento Frágil e Entrada Inválida, conforme a tabela I.

TABELA I: CLASSIFICAÇÃO DE VULNERABILIDADES

Classificação de Vulnerabilidades			
Classificação	***Risco***	***Código***	***Impacto***
Armazenamento Frágil	M2	getSharedPreferences / getPreferences	C, I
		getFilesDir + openFileOutput	
		getExternalStorageDirectory + openFileOutput	
	M4	getCacheDir + HttpResponseCache.install	C

Classificação de Vulnerabilidades			
Classificação	***Risco***	***Código***	***Impacto***
		getExternalCacheDir + HttpResponseCache.install	
	M9	removeAllCookies	C
		remove / removeAll	
	M5	getSharedPreferences / getPreferences	C, I, D
		getFilesDir + openFileOutput	
		getExternalStorageDirectory + openFileOutput	
Entrada Inválida		inputType="textPassword"	
	M7	Insert / insertOrThrow / insertWithOnConflict / delete / update / updateWithOnConflict / query / rawQuery / replace / replaceOrThrow	C, I, D
	M8	checkCallingOrSelfPermission	C, I

Por exemplo, a plataforma Android disponibiliza como recurso para armazenar dados localmente os arquivos de preferências. Este mecanismo, chamado de *SharedPreferences,* permite a escrita e leitura em arquivos específicos de uma aplicação através de combinações de chave-valor. Para isto o desenvolvedor pode utilizar os métodos *getSharedPreferences* ou *getPreferences.*

Quando este método é empregado sem atenção às questões

de segurança o risco *M2 - Insecure Data* Storage estará presente. M2 está relacionado com o armazenamento de dados do aplicativo no dispositivo móvel de forma insegura e sua principal característica é a perda ou exposição de informações. Geralmente ocorre sob a forma de exposição de dados sensíveis, como histórico de transações, nomes de usuários, senhas, tokens, dados de localização, informações pessoais e dados do dispositivo. Um ataque bem-sucedido ao sistema que explore esta vulnerabilidade comprometerá a confidencialidade e a integridade das informações.

Para que um app não possua uma brecha para *Insecure Data Storage,* o d*esenvolvedor* deverá ser alertado de que os recursos das classes *getSharedPreferences* e *getPreferences* deverão ser evitados e, caso seja necessário utilizá-los, deverá ser usado o modo de acesso privado *MODE_PRIVATE* com o uso adicional de um algoritmo de criptografia.

Outro risco identificado, que ilustra esta classificação, foi *M7 - Client Side Injection.* Na forma de injeção de SQL é caracterizada pela introdução de códigos maliciosos e malformados que, ao serem processados pelo sistema, atuarão em conjunto com a linguagem utilizada para manipulação de dados, formando novos comandos que serão executados pelo Sistema de Gerenciamento de Banco de Dados (SGBD). Possui como finalidade expor informações confidenciais ou manipular os dados do sistema. Neste sentido, considere que um atacante insira entradas maliciosas em uma Atividade de um aplicativo e que este não valide as entradas de dados recebidas antes de utilizá-las para processamento de uma consulta SQL. Quando o *app* processar a requisição, ele poderá obter ou alterar informações que ele não deveria acessar naquele contexto.

Para manipular os dados através de consultas SQL que serão realizadas no SGBD SQLite, a plataforma Android disponibiliza uma série de métodos para esta manipulação, a partir da classe *SQLiteDatabase*. Especificamente para a seleção e a consulta de informações a plataforma oferece o método query. Este método recebe como parâmetros o nome da tabela onde a consulta será realizada *(table)*, as colunas desta tabela que serão retornadas *(columns),* a restrição aplicada (selection), os valores utilizados para restringir o retorno, através do símbolo "?" *(selectionArgs)*, o agrupamento de dados *(groupBy)*, a restrição do agrupamento (having) e a ordem de apresentação dos dados *(orderBy)*.

A vulnerabilidade *Client Side Injection* se manifesta em um aplicativo quando os métodos para manipulação de dados são utilizados sem parâmetros, com a concatenação de *strings* na cláusula where. A figura 1 apresenta a assinatura do método query com um exemplo que utiliza a concatenação de strings ao invés do uso do parâmetro *selectionArgs*.

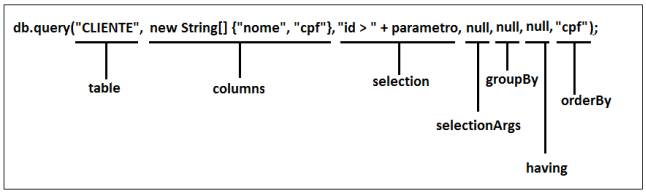

Fig. 1. Uma parte da lista de verificações do Lint

Para prevenir *Client Side Injection*, especificamente *SQL Injection* em banco de dados SQLite, o desenvolvedor deverá ser alertado para sempre validar os parâmetros de entrada. No que se refere aos métodos para consulta, caso o método query seja identificado e que os argumentos não estão sendo utilizados através do parâmetro *seleciontArgs*, que é responsável por não permitir que códigos maliciosos sejam inseridos, o desenvolvedor deverá ser alertado para o risco de que poderá sofrer um ataque deste tipo. Estas orientações preservam a confidencialidade, a integridade e a disponibilidade das informações do sistema.

A partir da identificação das vulnerabilidades, dos aspectos de segurança violados e da identificação das classes e métodos que podem permitir que estas vulnerabilidades sejam exploradas, estas foram agrupadas e classificadas quanto a exposição das informações e às formas de validação empregadas para prevenilas.

A categoria "Armazenamento Frágil" está relacionada com o armazenamento precário das informações. Este agrupamento considerou que as informações armazenadas estarão expostas no momento em que a vulnerabilidade for explorada, como por exemplo, dados armazenados em cache sem criptografia que podem ser obtidos diretamente por um atacante. Em contrapartida, "Entrada Inválida" é relativa às vulnerabilidades associadas com as informações fornecidas como entrada para processos internos do sistema. O que caracteriza este agrupamento é a exposição de uma informação a partir da utilização de dados de entrada sem validação.

Esta classificação fornece as definições necessárias para analisar, identificar e avaliar códigos potencialmente vulneráveis de um aplicativo, indicando qual o impacto nos aspectos da segurança da informação da confidencialidade (C), integridade (I) e disponibilidade (D) serão violados.

Baseado em [10], que definiu que a deteção antecipada de vulnerabilidades auxilia o desenvolvedor a se conscientizar dos problemas de segurança ocasionados por um código mal escrito, fazendo com que os mesmos não aconteçam de forma repetitiva e nesta classificação de vulnerabilidades, esta abordagem foca nos problemas de segurança identificados que ocorrem no nível do código-fonte da aplicação. Para que o objetivo enunciado seja atingido as técnicas de deteção antecipada de vulnerabilidades e de análise estática foram utilizadas.

Dentre as vantagens pela escolha em se utilizar a técnica de deteção antecipada está o fato de que desta forma não é necessário utilizar nenhum método adicional para obtenção do código-fonte a partir do bytecode do aplicativo, como por exemplo, realizar a engenharia reversa do código. Outras técnicas também podem ser usadas para dificultar a obtenção e análise do código-fonte, como a ofuscação do código, o que tornaria este processo ainda mais complicado.

Outra vantagem desta técnica encontra-se no fato de que detetar antecipadamente as vulnerabilidades tem como consequência o desenvolvimento do software seguro, haja vista que aquelas são identificadas, analisadas e corrigidas antes do aplicativo ser finalizado e publicado para uso.

A opção pela técnica de análise estática de código está relacionada ao fato de que esta é, geralmente, empregada para analisar o código-fonte de um sistema sem executá-lo. Isto normalmente é realizado através do uso de uma ferramenta que examina o código para encontrar erros ou vulnerabilidades durante o processo de codificação, permitindo que o desenvolvedor o corrija quando um problema é informado [14]. Por fim, a análise estática permite que todo o código-fonte seja analisado. Este tipo de análise pode revelar erros e vulnerabilidades antes que estes aconteçam ou sejam explorados.

III. FERRAMENTA DE SUPORTE

A ferramenta appDroidAnalyzer foi codificada a partir da criação de uma extensão da ferramenta Lint, que é distribuída em conjunto com a Android Development Tools (ADT), pela Google, a partir da versão 16. Lint utiliza a árvore sintática da linguagem de programação Java para percorrer o código-fonte, o que permite identificar as partes vulneráveis do código a partir da

técnica de análise estática de código com casamento de padrões. Lint analisa os arquivos que integram o projeto do aplicativo (arquivos Java, XML, ícones e configurações) à procura de oportunidades para otimização do código no que diz respeito à exatidão, segurança, desempenho, usabilidade, acessibilidade e internacionalização [15].

Como appDroidAnalyzer foi criada a partir de uma extensão do Lint, esta ferramenta foi desenvolvida através da implementação de classes Java, que foram registradas na biblioteca do Lint para que esta passasse a utilizar as novas verificações em sua rotina de análise do código-fonte. A figura 2 apresenta um trecho da lista de verificações desenvolvidas e registradas na base de pesquisa do Lint.

```
Available issues:
Security
========
SQLInjectionQueryDetector
-------------------------
Summary: Injeção de SQL detectada!
Priority: 5 / 10
Severity: Warning
Category: Security
Não utilize queries dinâmicas sem a utilização de parâmetros no SQLite.  Este tipo de código permite que um usuário
mal-intencionado obtenha informações sensíveis da aplicação sem que o mesmo possua autorização para isto.
More information:
https://www.owasp.org/index.php/Mobile_Top_10_2014-M7
```

Fig. 2. Uma parte da lista de verificações do Lint

As vulnerabilidades que foram identificadas e classificadas neste trabalho serviram para limitar a atuação desta ferramenta. Para cada tipo de código identificado uma nova classe foi implementada para detetar o problema. Estas classes implementam Detector.JavaScanner do Lint e analisam o código-fonte a procura de padrões de código que possam introduzir as vulnerabilidades. Quando uma possível vulnerabilidade é encontrada, ela é inserida no relatório de logs do Lint. Isto faz com que um alerta seja emitido para o desenvolvedor, contendo uma explicação do problema, o que pode ser feito para evitá-lo, a sua prioridade e sua categoria.

Uma vez que a ferramenta appDroidAnalyzer identifica, em tempo de codificação, os problemas de segurança utilizados neste trabalho, ela proporciona aos desenvolvedores a oportunidade de se obter uma rápida resposta em relação ao código que eles estão escrevendo. appDroidAnalyzer possui um grande potencial para ofertar ao desenvolvedor a chance de aprender a não cometer os mesmos erros em termos de segurança, uma vez que ele será constantemente notificado dos problemas encontrados no momento em que os mesmos forem identificados.

IV. AVALIAÇÃO

Para avaliar a técnica apresentada, inicialmente, o código-fonte de dois aplicativos foram submetidos à análise da ferramenta appDroidAnalyzer. A aplicação *WAPPushManager* é responsável por receber e processar as mensagens do tipo *WAPPush* e, até a versão 5.0 do Android, é vulnerável à injeção de SQL no banco de dados SQLite. Esta vulnerabilidade foi catalogada na CVE-2014-8507 e o *IBM X-Force* Exchange a classificou como como crítica e de fácil exploração.

Para explorar esta vulnerabilidade o atacante envia uma mensagem *WAPPush* malformada para o telefone da vítima com a finalidade de abrir uma Atividade ou executar um serviço. Quando esta mensagem é recebida, ela é processada pelo método *dispatchWapPdu*. Este método recebe o identificador da aplicação e o conteúdo da mensagem e repassa estas informações para o método *queryLastApp*, que é responsável por executar uma consulta no banco de dados com estes parâmetros e retornar as informações necessárias para a classe devolver o serviço solicitado.

Este método é vulnerável porque utiliza a concatenação de *strings* para montar a consulta *SQL* que será executada pelo módulo, conforme a figura 3, permitindo que os parâmetros passados pela mensagem malformada façam acesso à um recurso do sistema que o atacante pode não possuir permissão. Esta vulnerabilidade foi corrigida a partir da versão 5.0 do Android, onde o método *queryLastApp* deixou de usar a concatenação de *strings* e passou a utilizar uma consulta parametrizada.

Código Vulnerável	Código Corrigido e Seguro
String sql = "select install_order, " + " package_name, " + " class_name, app_type, " + " need_signature, further_processing" + " from " + APPID_TABLE_NAME + " where x_wap_application=\'" + app_id + "\'" + " and content_type=\'" + content_type + "\'" + " order by install_order desc"; Cursor cur = db.rawQuery(sql, null); 2 1	Cursor cur = db.query(APPID_TABLE_NAME, new String[] {"install_order", "package_name", "class_name", "app_type", "need_signature", "further_processing"}, "x_wap_application=? and content_type=?", new String[] {app_id, content_type}, null /* groupBy */, null /* having */, "install_order desc" /* orderBy */); 2 1

Fig. 3. Análise do código do método queryLastaApp

No lado esquerdo da figura 3 podemos ver que o item identificado pelo número 1 concatena as informações recebidas pelos parâmetros *app_id e contente_type* com a cláusula *where* da consulta, confiando na informação recebida, e processa-a através do comando rawQuery identificado pelo número 2. O ataque será consumado caso o conteúdo recebido seja malicioso. Já no lado direito da figura vemos que o problema foi tratado e resolvido. O método *rawQuery* foi substituído pelo método query, e não existe mais a concatenação de *strings* dos parâmetros recebidos com a cláusula *where* da consulta. Esta situação foi substituída pela utilização dos parâmetros fornecidos pelo método *query,* onde as informações recebidas são repassadas para a consulta deste.

A análise do módulo *WAPPushManager* evidenciou, conforme a figura 4, que ele foi considerado pela appDroidAnalyzer suscetível a injeção de SQL, vulnerabilidade classificada como "Entrada Inválida", o que pode comprometer a confidencialidade, a integridade ou a disponibilidade dos dados do sistema.

```
Scanning smspush: ...
src\com\android\smspush\WapPushManager.java:120: Warning: Evite consultas no
SQLite sem passar parametros para a query.
Este tipo de codigo permite que um usuario mal-intencionado, sem autorizacao,
obtenha dados privados, violando os aspectos de confidencialidade, integridade
ou disponibilidade. [SQLInjectionRawQueryDetector]
        Cursor cur = db.rawQuery(sql, null);
                     ~~~~~~~~~~~~~~~~~~~~~~
0 erros, 1 warnings
```

Fig. 4. Análise do WAPPushManager

Já o *app Stick Cricket*, que possui como característica armazenar informações do jogo no próprio dispositivo do usuário, ao ser analisado, foi considerado vulnerável quanto ao armazenamento dos dados na memória interna do dispositivo.

Uma das informações que este jogo mantém é a maior pontuação obtida pelo jogador. Para armazenar estas informações este aplicativo utiliza os recursos da classe *sharedPreferences*. A figura 5 apresenta a tela inicial do aplicativo com a pontuação obtida pelo usuário e, ao seu lado, o fragmento do arquivo de preferências com a identificação do trecho que armazena esta pontuação.

Como pôde ser visto na figura 5, este aplicativo não utiliza

nenhum método de criptografia para armazenar as informações de forma segura, logo, um usuário mal-intencionado pode alterar estas informações, como por exemplo, para aumentar a sua pontuação e obter alguma vantagem a partir disto.

A análise do código-fonte do app Stick Cricket demonstrou que este aplicativo é vulnerável quanto ao armazenamento dos dados na memória interna do dispositivo, violando as informações perante os aspectos da confidencialidade e da integridade, conforme figura 6.

Fig. 5. Pontuação do jogo Stick Cricket

```
Scanning StickCricket: .....
      - appcompat_v7: ..
src\com\sticksports\stickcricket\Cocos2dxHelper.java:238: Warning: Evite armazenar
informações utilizando getSharedPreferences sem criptografia.
Este tipo de codigo permite que um usuario mal-intencionado, sem autorizacao, obtenha
dados privados, violando os aspectos de confidencialidade, integridade  ou disponibilidade.
[GetSharedPreferencesDetector]
      return ((Activity) sContext).getSharedPreferences(PREFS_NAME, 0).getBoolean(key,
defaultValue);
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
0 errors, 10 warnings
```

Fig. 6. Trecho da análise do Stick Cricket

As vulnerabilidades contidas nestas aplicações poderiam ter sido evitadas se, durante a fase de codificação, uma técnica como a apresentada nesta pesquisa fosse utilizada para auxiliar o desenvolvedor a identificar e corrigir o código.

Após esta análise inicial foi necessário obter outros aplicativos para validar a eficácia desta técnica. Para isto foi necessário identificar e selecionar um repositório específico para aplicativos para a plataforma Android. Outra característica utilizada como restrição para esta seleção foi que a base de dados possuísse somente apps classificados como software livre e de código aberto. Respeitando estas premissas, o repositório F-Droid foi escolhido por atender a todos os requisitos definidos anteriormente. Este repositório é um banco de dados de aplicativos classificados como *Free and Open Source Software* (FOSS) [16].

Foram obtidos e submetidos à análise da ferramenta appDroidAnalyzer, em duas fases, 857 aplicativos deste repositório. Na primeira etapa buscou-se identificar quais aplicativos possuíam uma potencial vulnerabilidade em relação à classificação "Armazenamento Frágil" e foram encontrados 51 aplicativos suscetíveis a esta fragilidade. No segundo ciclo a ferramenta procurou identificar vulnerabilidades relacionadas com a classificação "Entrada Inválida". Neste caso, 10 aplicativos foram encontrados. De todos os aplicativos analisados, 2 apresentaram ser suscetíveis aos dois tipos de vulnerabilidade.

O estudo demonstrou que, no total, 65 aplicativos de todos os que foram submetidos à verificação apresentaram pelo menos um dos tipos de vulnerabilidades classificadas neste trabalho.

Somando-se à esta validação as duas avaliações inicias com o módulo *WAPPushManager* e com o *app Stick Cricket*, 52 aplicativos possuem "Armazenamento Frágil" e 13 possuem a vulnerabilidade "Entrada Inválida", conforme figura 7.

V. TRABALHOS RELACIONADOS

A. *Abordagem de deteção antecipada*

Souza [10] identificou dois problemas em relação a segurança de software: que as ferramentas existentes para validação de vulnerabilidades para aplicações web atuam somente no fim do processo de desenvolvimento, quando a aplicação está concluída; e a técnica de casamento de padrões de código, que é muito utilizada, gera um alto número de falsos positivos. Desta forma, o autor utilizou as premissas de que a deteção antecipada de vulnerabilidades ajudaria o desenvolvedor a melhorar a qualidade do seu código em termos de segurança e que a técnica de Análise de Fluxo de Dados (*Data Flow Analysis - DFA*), utilizada por compiladores, diminuiria a taxa de falsos positivos quando comparada com a utilização de casamento de padrões de código. Baseado na lista de vulnerabilidades *OWASP - Top Ten 2013* e na técnica de DFA ele desenvolveu um *plug-in* para a IDE Eclipse chamado *Early Security Vulnerability Detector* (ESVD).

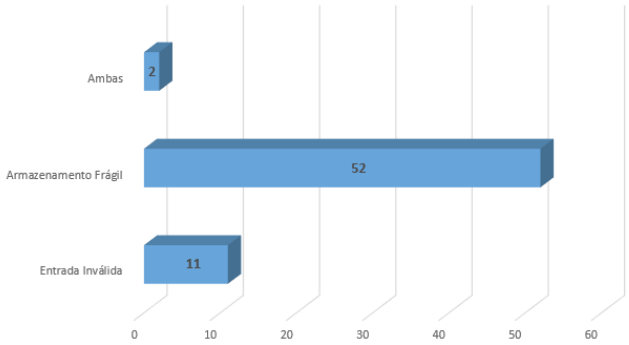

Fig. 7. Resultado apurado

Em uma primeira análise com seis projetos de software que foram analisados pela ESVD e por outras três ferramentas que implementam a técnica de casamento de padrões, o autor concluiu que a análise de fluxo de dados reduz significativamente a taxa de falsos positivos quando comparada com a outra técnica.

Em outro experimento com 27 participantes divididos em dois grupos, que teve como finalidade validar se a deteção antecipada ajudaria o desenvolvedor a corrigir as vulnerabilidades de um programa, Souza concluiu que a deteção antecipada ajuda o desenvolvedor a prevenir ou remover as vulnerabilidades encontradas no código.

Outra abordagem a partir da deteção antecipada de problemas foi utilizada por Saglam [11], que partiu da hipótese de que os apps que possuem problemas de lentidão ocasionados pelo mal gerenciamento da memória são mal avaliados na Google Play e, consequentemente, possuem uma baixa reputação. Esta classificação considerou com baixa reputação os aplicativos avaliados com uma ou duas estrelas e com boa reputação os que receberam quatro ou cinco estrelas. Os aplicativos com uma boa avaliação possuem em comum a rapidez para iniciar e abrir atividades, a estabilidade durante o uso (não congelar ou fechar inesperadamente) e apresentar rápida resposta a cada solicitação de um serviço pelo usuário.

O autor identificou quais os métodos Java para a plataforma Android estão relacionados com os problemas referentes ao gerenciamento ruim de memória e em seguida, classificou-os em quatro tipos: *Using non-static inner classes, Not setting thread priorities, Not using a cancellation policy in a thread e Not reusing views in list view.* Para identificar estas más-práticas, foi desenvolvida a ferramenta Bad-Practice Finder, baseada em Lint, que procura por códigos que possam ocasionar lentidão devido

aos problemas de gerenciamento de memória classificados em uma destas quarto categorias.

O autor obteve 932 aplicativos de um repositório *open-source* e comparou as informações destes com a sua reputação na Google Play. Os cem *apps* com as piores avaliações foram utilizados em sua análise através da ferramenta Bad-Practice Finder.

Após obter os resultados, [11] concluiu que a remoção de práticas ruins de codificação melhora a percepção dos usuários em relação à lentidão, estabilidade e baixo consumo de memória, e que esta melhora está diretamente relacionada com as avaliações que o *app* recebe na Google Play.

B. Abordagem de deteção tardia de vulnerabilidades

Em Cheng et al. [7] foi proposto um método para detetar comportamentos suspeitos em aplicações Android, que consistiu em analisar o dex bytecode dos aplicativos. Este processo simula a execução dos aplicativos e identifica cinco tipos diferentes de vulnerabilidades: *a) Service Provider services ordering; b) Privacy leak; c) Charges consumption; d) Native code; e) Constant URL connection.*

Para avaliar este método foi desenvolvido o protótipo ApkRiskAnalyzer, que extrai o dex bytecode do app do pacote da aplicação (*Android Package* - APK) que é armazenado em um Sistema Gerenciador de Banco de Dados (SGBD) MySQL. Em seguida é realizada a simulação dos processos com cada *dex bytecode* obtido e armazenado no SGBD. Esta simulação é realizada através de dois mecanismos. No mecanismo *Taint Analysis* é feita uma análise para identificar potenciais comportamentos perigosos a partir do uso incorreto de dados sensíveis. No mecanismo *Constant Analysis* é verificado como as informações são armazenadas em variáveis do tipo constante.

Finalmente, é realizada uma análise estática para comparar os comportamentos identificados anteriormente com uma base de dados de regras vulneráveis previamente cadastradas pelos autores. Quando é encontrado um casamento de padrões entre esta base de conhecimento e o comportamento inseguro, o risco é classificado em uma das cinco classificações anteriores.

Este método foi aplicado em 630 aplicações que foram obtidas na loja Google Play, e deste total, foram identificadas estas práticas em 575 *apps.*

Outra abordagem baseada no arquivo executável *dex bytecode* foi utilizada por Wu et al. [9], que define um comportamento chamado pelos autores de *Exposed Component Vulnerability* (ECV). Este comportamento permite que aplicações exponham componentes para utilização, de forma colaborativa, de outros aplicativos, o que torna possível que uma aplicação envie uma requisição ou uma entrada maliciosa para um outro componente disponível no sistema ou para outra aplicação.

Neste trabalho, [9] apresenta uma abordagem para identificar e classificar as vulnerabilidades relacionadas com vazamento de informações contidas em ECVs em Android. Para isto, foi realizada uma análise para criar uma classificação e um catálogo de regras de sintaxe baseadas em uma combinação de métricas (por exemplo, semântica de permissões e nomes de *Applications Programming Interface* - APIs). Este processo forneceu um conjunto de dados, que especifica um conjunto de ataques relacionados com as ECVs, chamado *Vulnerability-specific Sink (VSinks)*.

Após esta classificação, um algoritmo que realiza uma análise intra-procedural iterativa do aplicativo para criar mapeamento do controle de fluxo do aplicativo foi utilizado. Apoiado em uma análise deste fluxo a vulnerabilidade é classifica em uma das quatro categorias: *VS_Direct* (vulnerabilidades relacionadas com privilégios de acesso aos recursos do sistema); *VS_Public* (vazamento de informações, mas que não estão relacionadas diretamente com os privilégios do aplicativo); *VS_DirectByPerm* (similar à *VS_Direct,* porém neste caso serão considerados os parâmetros enviados para um método sob ataque); e *VS_Input* (agrupa os VSinks que fazem mau uso de recursos privilegiados). Para as ECVs classificadas em V*S_DirectByPerm e VS_Input* são removidos os falsos positivos a partir de uma análise semiguiada através dos parâmetros utilizados nos métodos considerados vulneráveis.

A ferramenta ECVDetector, criada pelos autores, implementou esta classificação e analisou 1.000 aplicativos, onde constatou-se que, após o experimento, 49 *apps* foram considerados vulneráveis.

Já em Kim et al. [8] foi desenvolvida a ferramenta ScanDal, que tem como a finalidade encontrar vazamento de informações em aplicativos Android e implementa a técnica de análise estática. Dois tipos de problemas foram utilizados em sua análise: as APIs que retornam informações sensíveis de recursos do sistema, como a localização física do usuário, identificadores do telefone (IMEI, número de série dentre outros) e dados de áudio e vídeo; e as APIs que transferem dados pela rede, como o envio de um arquivo através da internet ou o envio de um SMS.

Para isto, ScanDal analisa o aplicativo em três fases. Na fase *Front-end* é realizada a extração do *dex bytecode* obtido a partir do APK do aplicativo. Em seguida, na fase *Translation*, é realizada a conversão do arquivo executável para uma linguagem intermediária criada pelo autor, chamada de *Dalvik Core*. Finalmente, na fase *Abstract Interpretation*, ocorre a interpretação desta linguagem para que sejam identificados os trechos vulneráveis do código-fonte.

ScanDal detetou o vazamento de informações em 11 aplicativos de 90 *apps* gratuitos analisados e obtidos na loja Google Play. Destes, 6 enviavam dados de localização do usuário para servidores de propagandas, 5 armazenavam estas informações em arquivos e 1 enviava o IMEI do dispositivo para um servidor remoto. Em outra análise, 8 *apps* obtidos em loja de terceiros, que armazenam apps modificados, foram submetidos a ScanDal. Foi identificado o vazamento de informações em todos estes, como dados de localização e do dispositivo, como o IMEI.

C. Análise dos trabalhos relacionados

Primeiramente, o presente trabalhos se diferencia dos demais dado que nenhum outro define uma classificação que correlacione os problemas de segurança com os princípios definidos pela norma ABNT NBR ISO/IEC 27002:2013, confidencialidade, integridade e disponibilidade.

Além da classificação de vulnerabilidades proposta neste trabalho, a principal diferença entre a abordagem utilizada em [10] com a deste trabalho encontra-se na plataforma utilizada, onde [10] utilizou uma avaliação voltada para aplicações Web. Esta característica faz com que o seu conteúdo não seja aplicável à plataforma Android. A complexidade para avaliar um código para Android é diferente de validar um código Java para Web. Além disto, outra característica que diferencia o Android de Web é que a plataforma móvel permite ao desenvolvedor utilizar recursos do dispositivo, como o acesso de leitura e escrita a agenda de contatos. Outra característica é que em Android a manipulação de comportamentos e atributos são feitos com classes Java específicas para esta plataforma, a partir de um mapeamento da interface, criada em XML (E*xtensible Markup Language* - Linguagem de Marcação Extensível), para um objeto, e a partir deste objeto a linguagem Java é utilizada para manipular os atributos e comportamentos deste, com classes e

métodos específicos desta plataforma. Já na linguagem Java para desenvolvimento Web não existe este mapeamento anterior.

Em comparação com [11], ainda que aborde os problemas de forma antecipada e crie uma ferramenta a partir da extensão do Lint, este trabalho não se preocupou com o aspecto de segurança dos *apps*. O autor identifica problemas de desempenho relacionados com o gerenciamento de memória e a sua relação com a reputação do aplicativo nas lojas oficiais.

Em relação aos trabalhos [7], [8] e [9], a principal diferença reside no fato de que nestes a análise do aplicativo não é feita através do código-fonte, mas é realizada a partir do d*ex bytecode*, após este estar pronto e publicado em uma loja oficial. Nestes casos, não há a prevenção para evitar que potenciais vulnerabilidades sejam inseridas durante o processo de desenvolvimento do aplicativo, de forma antecipada, pois o código-fonte não é analisado durante esta fase. Estas comparações podem ser visualizadas na tabela II.

TABELA II: COMPARATIVO ENTRE TRABALHOS

	Plataforma Android	Detecção Antecipada	Evita vulnerabilidade	Correlata CID	Código-Fonte	Relacionado à segurança
Cheng et al.	✓	×	×	×	×	✓
Kim et al.	✓	×	×	×	×	✓
Saglam	✓	✓	✓	×	✓	×
Souza	×	✓	✓	×	✓	✓
Wu et al.	✓	×	×	×	×	✓
Ferreira et al.	✓	✓	✓	✓	✓	✓

Em termos numéricos, [7] identificou 91% de comportamentos suspeitos nos aplicativos analisados, enquanto que [9] identificou 4,9% apps vulneráveis, ao passo que [8] encontrou 12%. Na prova de conceito realizada por este trabalho 7,6% dos aplicativos submetidos à análise da appDroidAnalyzer possuíam vulnerabilidades. Em relação aos trabalhos de [10] e [11] não é possível identificar o percentual devido à finalidade de cada uma destas pesquisas. A tabela III apresenta esta comparação.

TABELA III : COMPARATIVO ENTRE TRABALHOS (%)

	Total	Encontrados	%
Cheng et al.	630	575	91%
Wu et al.	1000	49	4,9%
Kim et al.	90	11	12%
Saglam	-	-	-
Souza	-	-	-
Ferreira et al.	859	65	7,6%

VI. CONCLUSÕES

A pesquisa e identificação de vulnerabilidades em aplicativos para a plataforma Android tem utilizado uma abordagem baseada na identificação tardia, quando os *apps* já estão publicados para *download* em lojas e milhares de usuários já podem ter sido afetados. Visto que 90% destes *apps* possuem vulnerabilidades, é de grande importância que estas sejam identificadas durante o processo de desenvolvimento, enquanto o desenvolvedor está codificando o *app*.

A técnica proposta neste trabalho permite ao desenvolvedor conhecer as vulnerabilidades, os tipos de código que permitem a introdução destas e a sua relação com os atributos de segurança da informação confidencialidade, integridade e disponibilidade. Ao utilizar esta técnica, o desenvolvedor terá a oportunidade de identificar, analisar e corrigir o código-fonte do aplicativo, antecipadamente, para evitar que as vulnerabilidades sejam inseridas no *app*, preservando aqueles atributos de segurança.

Este estudo demonstrou que, no total, 65 aplicativos de todos os que foram submetidos à verificação da ferramenta desenvolvida apresentaram pelo menos um tipo de vulnerabilidade das que foram classificadas nesta pesquisa. Destes, 52 *apps* são suscetíveis à "Armazenamento Frágil" enquanto que em outros 11 as vulnerabilidades classificadas como "Entrada Inválida" foram identificadas. Finalmente, de todos os aplicativos analisados, 2 apresentaram vulnerabilidades relativas às duas classificações. Isto demonstra que identificar, apoiado em uma ferramenta, as fragilidades de um app, antecipadamente, em tempo de codificação, possibilita ao desenvolvedor evitá-las, uma vez que, naquela oportunidade, ele possui o contexto do código que ele está desenvolvendo.

REFERENCES

[1] Chau, Melissa and Reith, Ryan and Govindaraj, Navina and Nagamine, Kathy, "Smartphone OS Market Share, 2016 Q3". International Data Corporation, 2016. Disponível em: <http://www.idc.com/promo/smartphone-market-share/os>. Acessado em: 05 jan. 2017.

[2] AppBrain, "Android Statistics - Number of Android applications". 2017. Disponível em: <http://www.appbrain.com/stats/number-of-android-apps>. Acessado em: 12 fev. 2017.

[3] ARXAN Technologies, "5th Annual State of Application Security Report". 2016. Disponível em: <https://www.arxan.com/wp-content/uploads/2016/01/State_of_Application_Security_2016_Consolidated_Report.pdf>. Acessado em: 12 jan. 2017.

[4] OWASP, Open Web Application Security Project, "Mobile Security Project". Disponível em: <https://www.owasp.org/index.php/OWASP_-Mobile_Security_Project>. Acessado em: 09 nov. 2014.

[5] Dantas, Marcus Leal, "Information Security: an approach focused on risk management.", "Segurança da Informação: uma abordagem focada em gestão de riscos." Recife: Livro Rápido-Elógica. 2011.

[6] Crespo, María Augusta and Chóez, Rosalía Eulogia Ramos, "Study of the financial impact of the vulnerabilities of the websites of the banks in Ecuador", "Estudio del impacto financiero de las vulnerabilidades de las páginas web de los Bancos en Ecuador". Universidad Politécnica Salesiana. 2012.

[7] Cheng, Shaoyin and Luo, Shengmei and Li, Zifeng and Wang, Wei and Wu, Yan and Jiang, Fan, "Static detection of dangerous behaviors in android apps". Cyberspace Safety and Security. Springer International Publishing, 2013. p. 363-376.

[8] Kim, Jinyung and Yoon, Yongho and Yi, Kwangkeun and Shin, Junbum and Center, SWRD, "ScanDal: Static analyzer for detecting privacy leaks in android applications". MoST, 2012, 12.

[9] Wu, Daoyuan and Luo, Xiapu and Chang, Rocky KC, "A Sink-driven Approach to Detecting Exposed Component Vulnerabilities in Android Apps". Hong Kong, The Hong Kong Polytechnic University, 2014, arXiv preprint arXiv:1405.6282.

[10] Souza, Luciano Sampaio Martins de, "Early Vulnerability Detection for Supporting Secure Programming". Rio de Janeiro, Pontifícia Universidade Católica do Rio de Janeiro, 2015.

[11] Saglam, Ġsmagl Alper, "Measuring and Assesment of Well known Bad Practices in Android Application Developments". Middle East Technical University, 2014.

[12] Albuquerque, Ricardo and Ribeiro, Bruno, "Segurança no Desenvolvimento de Software - Como desenvolver sistemas seguros e avaliar a segurança de aplicaçes desenvolvidas com base na ISO 15.408", in Editora Campus. Rio de Janeiro, 2002

[13] ASSOCIAÇÃO Brasileira de Normas Técnicas, "ABNT ISO/IEC 27002 - Information Technology - Security Techniques - Code of Practice for Information Security Management, 2013.", "ABNT ISO/IEC 27002 - Tecnologia da Informação - Técnicas de Segurança - Código de Prática para Controles de Segurança da Informação, 2013".

[14] Li, Li and Bissyande, Tegawendé, François D Assise and Papadakis, Mike and Rasthofer, Siegfried and Bartel, Alexandre and Octeau, Damien and Klein, Jacques and Le Traon, Yves, "Static analysis of android apps: A systematic literature review". SnT, 2016.

[15] Developers, Android, "Improve Your Code with Lint", Disppnível em: https://developer.android.com/studio/write/lint.html. Acessado em: 08 out. 2015.

[16] F-Droid,"Category:Apps".F-Droid,2013.Disponívelem:<https://f-droid.org/wiki/page/Category:Apps>.Acessado em: 11 jun. 2016.