INFORMATION TECHNOLOGY

UDC 004.056.52/004.432(045)

Vasyl Melnyk[1]
Katerina Melnyk[2]
Oksana Zhyharevych[3]

STATIC ANALYSIS OF SOURCE CODE MODELED FOR JAVA-PROGRAMS CONTAINING APPLICATIONS WITH ANDROID SECURITY

Lutsk National Technical University
75, Lvivska st., Lutsk, 43018, Ukraine
E-mails:[1]melnyk_v_m@yahoo.com; [2]ekaterinamelnik@gmail.com; [3]oz_lutsk@mail.ru

Abstract. *A static analysis techniques were combined with model-based deductive verification using solvers of the static model theory (SMT) to create a framework that, given an aspect of analysis of the source code, automatically generated with an analyzer outputting a conclusion information about this aspect. The analyzer is generated by translating of a program collecting semantic to outlined formula in first order over a few multiple submitted theories. The underscore can be looked as some set of holes or contexts corresponding to the uninterpreted APIs invoked in the program. As the program makes an import of the packages and uses classes' methods of these packages, it is importing the semantics of API invocations in first order assertion. The analyzer is using these assertions as models and their first logic order formula incorporates the specification behavior (its negation) of the described programs. A solver of SMT-LIB formula is treated as the combined formula for "constrain" and "solve" it. The "solved" form can be used for logic errors (security) identification Android-based Java-programs. The properties of Android security are represented as constraint and analysis aims to show the respecting for these constraints.*

Keywords: Android security; Java-program; static analysis; static model theory; program code.

1. Introduction

Software systems stencily manage mission-critical activities in organizations which rely on dependable, situation award and delivered in time classified or sensitive information. Information streams in such organizations are usually machined by common-built, with open source code, and traditionally cooperated security software. Such programs may be unlicensed with malicious code or vulnerability containing, which can be used for insider or outsider to seize a confidential data, reclassify documents, make some destroyed actions, and modify valuable information. So, the reliability and correct work of described above software to drive these systems have become important issues for today.

Program-independed errors in software systems like overflows of buffer or null dereferences can be used by malicious applications to make unsecured holes, through which unsecured confidential data can be captured. Most of these bugs are detected too late, when destructive effects are already appeared [1] complicating the task of runtime preceding mechanisms of fault handling for ensuring data recovery. A more important is the lack of peculiar tools for logical dependencies of program-depended errors detecting in applications. A testing of a well-known incident list resulting from software glitches opens that logical application-depended errors were the causes in [2,3] works. A lot of these logical faults were hard and difficult to reveal (opposite to simple errors) with using observed testing methods [4] alone.

In this paper we combine the static analysis techniques with model-based deductive verification using solvers of the static model theory (SMT) to create a framework that, given an aspect of analysis of the source code, automatically generated with an analyzer outputting the conclusion information about this aspect. The analyzer is generated by program translation for collecting semantic to outlined formula in first logic order over a few multiple submitted theories. The underscore can be looked as some set of holes or contexts corresponding to the uninterpreted APIs invoked in the program. As the program makes an import of the packages and uses classes' methods of those packages, it is importing the semantics of API invocations in first order assertion. The analyzer is using these assertions as models and their first logic order formula incorporates the specification behavior (its negation) of the described programs. A solver of SMT-LIB formula is treated as the combined formula for "constrain" and "solve" it. The "solved" form can be used for logic errors (security) identification of Android-based Java-programs. The properties of Android security are represented as constraint and analysis aims to show the respecting for these constraints

2. Related works

Software verification and validation techniques for software can be divided in three categories. The first one includes informal methods such as testing and monitoring. These techniques are scaling well by the most used technique in practice to validate software systems. The testing accounts for forty to sixty percent for efforts of developments [4, 5]. The traditional methods for software program testing [6], however, do not allow for formal specification and verification of high logical properties that need to satisfy the system. In the area of security critical software where exponential goes up in the number of possible situations to be dealt is inevitable, traditional testing methods can be with difficulty used to provide some confidence level. The second category of methods for software verification and validation includes traditional formal methods such as verification model and providing of theorems are sometimes too heavy and rarely can be used in practice without considerable manual work.

Checking of the model includes an automatic approach for verification more successful while dealing with finite state systems. It suffers not only from infamous state explosions problem but it also need the model performance for program software.

Third method category for program software verification and validation is settled on static analysis techniques and abstract interpretation [7,8]. The static analysis refers to automatic displaying behavior technique for programs during the time of compiling. While static analysis tools have touched with a great practical success, and have been integrated with state of the newer compilers, such tools can reveal only small and simple errors due to the lack of their deductive strength. So, traditional tools of static analysis cannot reveal a deadlock presence or the violation of common exclusion in concurred programs. The abstract interpretation is a technique for the program semantic collection, comparison and combination. It successfully has been used for inferring of program runtime properties that may be used for program optimization. On the next the most successful approaches for program analysis are reviewed. In the last years a lot of work has been made in the area of software static analysis. Some tools of static analysis, such as described in articles [9-14] are making lightweight analysis of data flow. In the [15] conference article the data flow analysis is provided for verification, described in "metalanguage", designed for checking ″automata″ encoding. Astree is a static program analyzer directed to confirm the runtime error absence in the installed programs, and can handle only "safe" C-subset rather than all C-language. It concerns only for separate runtime errors but general program properties. In [16] the linear relation analysis is used to reveal invariant linear inequalities numerical program variables. Their methods have been used for verify (analyze delays) in synchronous programs written in "Lustre-language". A few technique approaches have been considered to provide an approximate answer to the validation problem as enlargement, convex approximations and factoring of Cartesian [17]. These approximations are realized with using of part of polyhedral library. In the article [18] the predicate abstraction is used to analyze hybrid systems. In this method the finite abstraction of hybrid automated system is creating with priory using initial predicates taken of the user.

3. Android platform

Android is the software to develop mobile devices. It includes an operating system, key applications and middle-connection software. The main feature of the Android is that an application can use elements of other programs. To rich this, the system has to start the process when any of the other programs part is need. Android has no any single point of entrance like function main(). They have essential components, which the system can instantiate and start if necessary. There are four main component types: activity, services, audio receivers and connect providers.

Android has an aim to active the first three components. For activities and services, the intent is the pair: <action_name, data>, to display the previous action, which receiver has to except and data to process. In the framework of the represented program analysis this aimed object is keyword used to make data flow analysis and call of analysis function.

Android-architecture provides created applications with phone functions and protects users to minimize mistake consequences and harmful software. As it was said, in Android an application may exchange its data and functionality with other software, and these accesses have to be controlled carefully to provide the security. Android permissions are rights given to programs to perform such functions like photocopy, GPS using and phone call making. When applications are installing, they accept the unique identifier UID, and every application always runs under its name with the particle appointed device. Application UID is using for protection its data exchange with other applications.

4. Model architecture

Fig. 1 shows architecture of static analysis proposed on the model basis. The abstract collecting semantic of Java-program are represented as "marked" coercions. "Marked" can be considered as a set of holes and contexts corresponding to no interpreted APIs, i.e. API from library with unknown semantics. As program imports the packages and uses classes' methods in imported packages, the semantics of API invocations are importing here as first order assertions (constraints). These assertions are models used to "unmark" constraints of the abstract collecting semantics i.e. for "fill in" uninterpreted APIs "holes". Aspects of analysis are specifying as constraints.

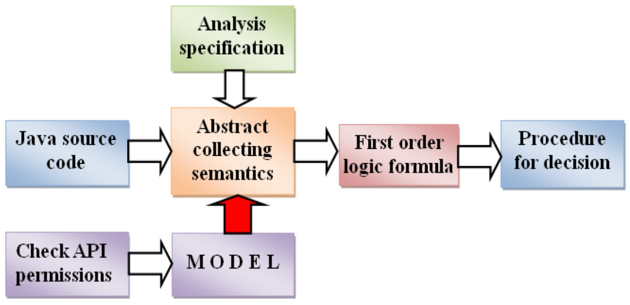

Fig. 1. Model architecture

The solving of the basic constraint is done by using of decision procedures combination provided by (Yices) constraint solver [19]. The key steps involved in the workframe of the analysis are 1) permission verification of the Android APIs, invoked in the Java code, based on Manifest.xml. The verification results of permission are using for model APIs modification, 2) generate abstract collecting semantics constraints from the Java code, 3) import models of uninterpreted methods and objects as assertions into already generated constraints; uninterpreted methods and/or objects have to be annotated by the developer; annotation is necessary from the moment since the particular method can be changed of the programmer and importing its "conventional" model from a model library may result the analysis unsoundness, 4) generate an analyzer by adding "aspect" constraints of the appropriate analysis, 5) Analyze by the constraint solving.

A. Verification of the permission

Android permissions can be separated into different security levels [20, 21], which are defined in the Mainifest.xml file. The framework examines this file and reveals permission information. The necessary list of permissions should be formed from analyzing of APIs invoked in the program and can be compared with permission information P taken from file Mainifest.xml. The android API calls can be mapped with necessary permission list used in [22]. If the every permission is provided we conclude that application has no permission violations. Otherwise, API calls which have no provided appropriate permissions will be considered as with return of -1 in the following analysis.

B. Constraints, solvers of SMT-LIB formula, satisfiability

Constraints are special formulas in first logic order [23], and a constraint system itself specifies the syntax and constraints semantics. The solver of the constraint implements a checking algorithm to verify the satisfiability/consistency of constraint sets using constraint theory, i.e., revealing if there exists a variable assignment that satisfies the constraints. A solver uses constraint theory axioms together with simplification rules as rewritten rules for the constraints transforming to a normal form which is called the "solved" form. The last constraint which results from the computation will be named the answer.

C. SMT-LIB formulas and Yices

Satisfiability Modulo Theories (SMT) libraries (SMT-LIB) [24] provide a framework for the first order formulas satisfiability checking in agreement with some background logical theories. SMT provides background theories standard description used in SMT systems. It gives a common input and output languages for SMT formula solvers. "Yices" described in [19] is effective SMT-LIB formula solver, that decides the arbitrary formulas satisfiability containing uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive data types, tipples, records, extensional arrays, fixed bit-vectors, lambda expressions, and quantifiers.

5. Java programs inferring collecting semantics

We have to perform intraprocedural and interprocedural analysis to analyze deep logical properties of Java-based programs. Using data flow analysis of the source code in intraprocedural analysis, in a series of steps is built a constraint system for capturing its collecting semantics. In the way of the interprocedural analysis the call graph has to be built and some external rules have to be defined for relating the different API invocations and detecting if the analyzed code breaks these rules.

In the *analysis* framework the following *sequence of steps* (SSA) has to be followed to check if the program satisfies an analysis aspect defined of the user. 1) The dataflow analysis of the Java source code has to be performed and its collecting semantic has to be generated, 2) The static single assignment [25] graph of the program, based on the dataflow analysis results, has to be generated, 3) The SSA graph has to be convert to the SMT-LIB formulas (see below), 4) On the last step the models of uninterpreted API invocations have to be imported as first order assertions.

The above steps can be illustrated in the following code:

```
1 class udhpcd
2 {
3      int getSocket (int listen_mode)
       {
5           int fd = 0;
6           if(listen_mode == 2) {
8               fd = listen_socket();
9           }
10          else {
12              fd = raw_socket() ;
13          }
14     }
15 }
```

In the program shown above listen_mode is the input of the user, listen_socket() is a return method with positive integer, raw_socket() is a method, providing of operated system, due to return specific positive integer. Provided data flow analysis of the source code is shown in fig 2. The integer number in the graph of the data flow indicates the code line number of each statement.

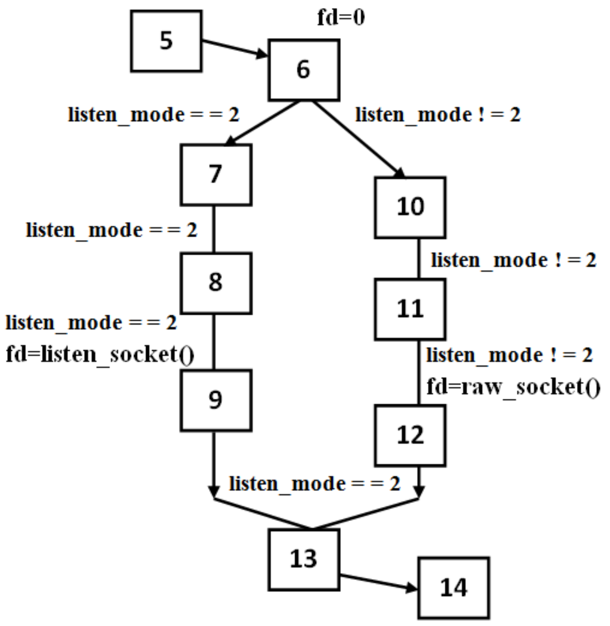

Fig. 2. Flow of Data

After line 12 the variable value fd can be as return value of listen_socket() or raw_socket(), because the above program has two branches. During the compilation time we can't determine, what way the control will pass. So, we consider that fd value is { listen_mode = 2 ∧ fd = listen_socket(); listen_mode ≠ 2 ∧ fd = raw_socket()}, where the semicolon present disjunction. To build SMT-LIB logic formulas for capturing the collection program semantics, we have to convert the program to static single assignment graph as shown in fig. 3.

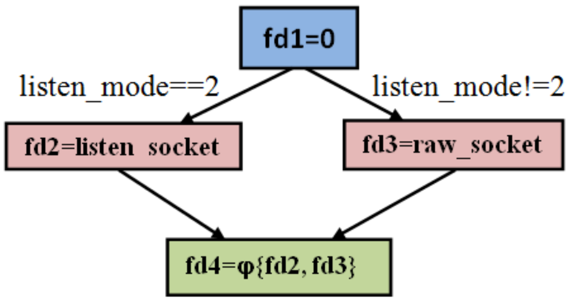

Fig. 3. Static Single Assignment for the program

From SSA graph shown on fig. 3, we make an assertion for every each. If first graph fd1 = 0, we can create (assert (= fd1 0)). The node "fd4 = Phi { fd3, fd2 }" is a φ-function, we built a disjunction (or (= fd4 fd3) (= fd4 fd2)). Here are two labeled edges, so, two implications have to be formed: (=> (= listen_mode 2)(= fd2 listen_socket)) and (=> (distinct listen_mode 2) (= fd3 raw_socket)). The APIs semantics are incorporated and if API has no permission provided we confirm that API returns -1. Else an assertion, characterizing API from site of the model library is imported. For example, for listen_socket and raw_socket, the assertion (assert (and (> listen_socket 1) (= raw_socket 1))) is imported.

The main program condition considered above is fd > 1. Verifying, if this condition holds is considered an aspect of the analysis for this program, which is included SMT-LIB formula, to characterize the collecting semantics of the program by adding the conjunct (< fd4 1). The combined SMT-LIB formula was unsatisfyable by the solver in [19], which indicates that the program satisfies the specification.

Now, the algorithm for SSA graph program converting to SMT-LIB formulas that captures it's collecting semantics, will be described down. Let G = ⟨N, ε⟩ be the SSA program graph. In this graph, every node represents a statement in the program. The if and loop conditions can be represented here as an edge graph labels, and SMT-LIB formulas can be generated to capture the program collecting semantics using algorithm 1 that formalizes the described above intuition.

Algorithm 1 to convert SSA algorithm into SMT algorithm

```
for $n \in$ N do
    if $n$ is a simple assignment statement VAR = EXP
    then
        Create an assertion (assert (= VAR EXP)) in
                                            SMT;
    end if
        if $n$ is a assignment statement with API call
                                    VAR =API
    then
        Create an assertion (assert (= VAR API));
    end if
    if $n$ is a $\varphi$ function statement then
        Let $v$ be the variable in this statement and
                the set W be
                the values of this $\varphi$
                function;
        Create a disjunction $v = wi$, where $wi \in$ W;
    end if
        if $n$ is a function call statement FUN()
    then
        Create an assertion (assert (= FUN
                            FUN_SUMMARY));

    end if
for $e \in$ E do
    if $e$ is labeled then
        Let $n$ be the node directed by this edge;
        Create a conjunction of implication
                formula $e \to n$;
    end if
end for
if The API permission is provided
then
    Provide the API model as (assert (= API
            API_specification value));
else
    Set the API model as -1 (assert (= API -1);
end if
Provide the function summary as the function
        return value after the function
        analyzed (assert (=FUN SUMMARY
        FUN return value)).
end for
```

6. Experiments and limitations

The source code from Android Bluetooth ChatServices application was analyzed above. This program builds a Bluetooth network platform to allow for device to exchange data with other devices of Bluetooth. It has three main functionalities: 1) searching of Bluetooth devices, 2) pair and connect the devices 3) transfer data between these devices. The application uses such API calls as:

BluetoothAdapter.getDefaultAdapter(),
BluetoothAdapter.getRemoteDevice(address),
BluetoothSocket and
BluetoothChatService.

These API models are provided here which are based on the Android Development Documentation. For this application we simply consider, that application can set up the Bluetooth service and can connect to any discovered devices. So, we set up BluetoothChatService to return non-null object, and in the SMT-specification we model the value of API-function as not returned -1. The source code analyzed satisfied specification. For this, a few Android-free source codes of the applications were been downloaded and run the static analysis tool to the source code described above. Some possible program vulnerabilities were been detected from the analysis, are under-mentioned.

Android SMSPopup: 1) in class SmsReceiverService.java there is a false null checker for statement. The branch if(message.isSms() && message.getMessageClass() == MessageClass.CLASS 0) will be never reached,

2) in class SmsPopupUtils.java the method getUnreadSmsCount(Context context) will be never called, 3) there is a system command call Runtime.getRuntime().exec(commandLine.toArray(new String [0])).getInputStream()), this statement may provide a command injection error.

openGPStracker: 1) in class Constants.java on line 98, there is a hardcoded password, 2) The function serializeWaypoints() in GpxCreator.java fails to perform a null checker for variable mediaUri (line 440), 3) The expression if(startImmidiatly && mLoggingState == Constants.STOPPED) (line 563) in GPSLoggerService.java is evaluated as true every time, the branch else will be never reached.

OpenSudoku: 1) method update() in IMNumpad.java (line 208 and 229) fails to perform a null checker for statement, 2) the method saveToFile() in FileExportTask. Returns of Java in a catch block (line 156) may lead to a lost error of the return value.

In the application Android SMSPopup the command injection error has been detected. The command statement is an array that comes from other function which possibly providing a wrong statement. In openGPStracker it is revealed more permissions than necessary; that may lead to the problem of overprivileged permission. There are also detected that the application openGPStracker is with hardcoded password.

A. Limitations

In the intraprocedural analysis, the constraint system includes all the possible variable values and this may result in false positives. There are necessary more sharp abstract interpretation methods to provide some more precise analysis. In the interprocedural analysis the summary based functions are modeled and this abstraction loses precision and gives out false negatives. Another problem of described above analysis tool is that it needs developers for external XML files creation to specify the right program properties; these files are difficult to build.

7. Conclusions

All Android programs can communicate to each other through system provided mechanisms such as files, activities, services, broadcast receivers and providers for communications. If developers use one of these techniques they have to be sure, that they communicate with the right entity, because permissions can be easily violate here inadvertently. The analysis program framework described in this paper can help developers to reveal programming errors and the statistic of the permission violations. Developers have to understand what permissions have to be set up correctly and what basic knowledge has to be need in the logic and limited solving. The task to help users to set up easily constraints and logical errors in programs can be accepted for the future work.

References

[1] *Yu W. D.* A software fault prevention approach in coding and root cause analysis. Bell Labs Technical Journal. 1998. Vol. 3, no. 2, pp. 3–21 [Online]. Available: http://dx.doi.org/10.1002/bltj.2101

[2] Software horror stories: http://www.cs.tau.ac.il/~nachumd/verify/horror.html.

[3] Forum on risks to the public in computers and related systems: http://catless.ncl.ac.uk/Risks/19.88.html.

[4] *Beizer B.* Software testing techniques (2-nd ed.). New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[5] *Woldman K. I.* A dual programming approach to software testing. Master's thesis, Santa Clara University, 1992.

[6] *Collard J.-F. Burnstein I* Practical Software Testing. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002.

[7] *Nielson F., Nielson H.R., C. Hankin* Principles of Program Analysis. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[8] *Cousot P., Cousot R.* "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." in Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ser. POPL '77. New York, NY, USA: ACM, 1977. – P. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[9] *Holzmann G. J.* "Software analysis and model checking," in CAV, 2002. – P. 1–16.

[10] *Evans D., Guttag J., Horning J., and Tan Y.*, "Lclint: A tool for using specifications to check code." in ACM SIGSOFT Software Engineering Notes. vol. 19, no. 5. ACM, 1994. – P. 87–96.

[11] *Anderson P., Reps T.W., Teitelbaum T., Zarins M.* «Tool support for fine-grained software inspection.» IEEE Software, vol. 20, no. 4. 2003. – P. 42–50.

[12] Evans D., Guttag J., Horning J., and Y. Tan, "Lclint: A tool for using specifications to check code." in ACM SIGSOFT Software Engineering Notes. vol. 19, no. 5. ACM, 1994. – P. 87–96.

[13] *Das M., Lerner S., Seigle M.* “Esp: Path-sensitive program verification in polynomial time.” in PLDI, 2002. – P. 57–68.

[14] *Martin F.* “PAG – an efficient program analyzer generator.” International Journal on Software Tools for Technology Transfer. vol. 2, no. 1, 1998. – P. 46–67.

[15] *Hallem S., Chelf B., Xie Y., Engler D.* «A system and language for building system-specific, static analyses.» In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. ACM Press, 2002. – P. 69–82.

[16] *Halbwachs N., Proy Y.-E., Roumanoff P.* “Verification of real-time systems using linear relation analysis.” in FORMAL METHODS IN SYSTEM DESIGN. 1997. – P. 157–185.

[17] *Halbwachs N., Merchat D., Parentvigouroux C.* “Cartesian factoring of polyhedra in linear relation analysis.” In Static Analysis Symposium. SAS03. Springer Verlag, 2003. – P. 355–365.

[18] *Alur R., Dang T., Ivancic F.* “Counterexample-guided predicate abstraction of hybrid systems.” Theor. Comput. Sci., vol. 354, no. 2, 2006. – P. 250-271.

[19] *Dutertre B., Moura L. D.* “The yices smt solver.” Tech. Rep., 2006.

[20] *Burns J.* “Developing Secure Mobile Applications for Android: An Introduction to Making Secure Android Applications.” iSec Partners, Tech. Rep., Oct. [Online]. Available: http://www.isecpartners.com/files/iSEC\Securing\ Android\Apps.pdf

[21] *Shin W., Kiyomoto S., Fukushima K., and Tanaka T.* “Towards formal analysis of the permission-based security model for android.” in Proceedings of the 2009 Fifth International Conference on Wireless and Mobile Communications. ser. ICWMC ’09. Washington, DC, USA: IEEE Computer Society, 2009. – P. 87–92. [Online]. Available: http://dx.doi.org/10.1109/ICWMC.2009.21

[22] *S.H.D.S. Adrienne Porter Felt, Erika Chin, D. Wagner* “Android permissions demystified.” in ACM Conference on Computer and Communication Security. 2011.

[23] *Frhwirth T., Abdennadher S.* “Principles of constraint systems and constraint solvers.” 2005.

[24] *Barrett C., Stump A., Tinelli C.* “The Satisfiability Modulo Theories Library (SMT-LIB).” www.SMT-LIB.org, 2010.

[25] *Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K.* “Efficiently computing static single assignment form and the control dependence graph.” ACM transactions on programming languages and systems. Vol. 13, 1991. – P. 451–490.

Received 25 November 2014.

В. М. Мельник[1], К. В. Мельник[2], О. К. Жигаревич[3]. Статистичний аналіз вихідного коду змодельований для java-програм, які містять додатки з безпекою Android

[1,2,3]Луцький національний технічний університет, вул. Львівська, 75, Луцьк, 43018, Україна
E-mails: [1]melnyk_v_m@yahoo.com; [2]ekaterinamelnik@gmail.com; [3]oz_lutsk@mail.ru

Здійснено поєднання методів статичного аналізу з моделлю дедуктивної перевірки й використанням рішень теорії статичної моделі (ТСМ) для створення основи, яка, враховуючи аспект аналізу вихідного коду, автоматично створюється за допомогою аналізатора, котрий виводить кінцеву інформацію про цей аспект. Аналізатор генерується шляхом перекладу програми для збору семантики з метою викладення формул в першому наближенні на основі кількох представлених теорій. Оскільки програма здійснює імпорт пакетів і використовує класові методи цих пакетів, вона імпортує семантику викликів API в наближенні першого порядку. Аналізатор, використовуючи ці наближення як моделі та їх формули першого порядку, залучає поведінку специфікації (його негативність) описаної програми. Рішення SMT-LIB формул розглядається як комбінована формула для того, щоб їх «обмежувати» та «розв’язувати». Форма «розв’язку» може використовуватися для ідентифікації логічних помилок (безпеки) Java-програм на базі Android. Властивості безпеки Android представлено як обмежувальні аналітичні цілі, щоб показати важливість цих обмежень.

Ключові слова: Android-безпека; Java-програма; програмний код; статичний аналіз; статична теорія моделювання

В. М. Мельник[1], К. В. Мельник[2], О. К. Жигаревич[3]. Статистический анализ исходного кодо смоделированный для java-программ содержащих приложения с безопасностью android

[1,2,3]Луцький національний технічний університет, вул. Львівська, 75, Луцьк, 43018, Україна
E-mails:[1]melnyk_v_m@yahoo.com; [2]ekaterinamelnik@gmail.com; [3]oz_lutsk@mail.ru

Проведено сопоставление методов статического анализа с моделью дедуктивной проверки и использования решений теории статической модели (ТСМ) для создания основания, которая, учитывая аспект анализа исходного кода, автоматически создается с помощью анализатора, выводящего конечную информацию об этом аспекте. Анализатор генерируется путем перевода программы для сбора семантики с целью изложения формул в первом приближении на основании нескольких представленных теорий. Так как программа делает импорт пакетов и использует классовые методы этих пакетов, она импортирует семантику вызовов API в приближении первого порядка. Анализатор, используя эти приближения как модели та их формулы первого порядка, включает поведение спецификации (его отрицательность) описанной программы. Решения SMT-LIB формул рассматривается как скомбинирована формула для того, чтобы их «ограничивать» и «решать». Форма «решения» может использоваться для идентификации логических ошибок (безопасности) Java-программ на базе Android. Свойства безопасности Android представлены как ограничивающие аналитические цели, чтобы показать важность этих ограничений.

Ключевые слова: Android-безопасность; Java-программа; программный код; статический анализ; статическая теория моделирования

Melnyk Vasyl. PhD in physics and mathematics. Assistant professor.
Computer Engineering Department, Lutsk National Technical University, Lutsk, Ukraine.
Education: Prekarpatian Stephanyk University, Ivano-Frankivsk, Ukraine (1995).
Research area: computing, programming and sockets.
Publications: 34.
E-mail: melnyk_v_m@yahoo.com

Melnyk Kateryna. PhD in technique. Assistant professor,
Computer Engineering Department, Lutsk National Technical University, Lutsk, Ukraine.
Education: Lesya Ukrainka Eastern European National University, Lutsk, Ukraine (1998).
Research area: computational intelligence systems.
Publications: 29.
E-mail: ekaterinamelnik@gmail.com

Zhyharevych Oksana. Assistant professor.
Computer Engineering Department, Lutsk National Technical University, Lutsk, Ukraine.
Education: Lutsk Biotechnical Institute of International Science and Technology, Lutsk, Ukraine.
Research area: computer programming, simulation-based semantics.
Publications: 20.
E-mail: oz_lutsk@mail.ru