ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com
International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 3, Special Issue 20, April 2016

INFORMATION FLOW CONTROLLER AND SECURITY POLICIES ENFORCER FOR MOBILE APPLICATIONS

S.N.ANITHA , K. KALYANI
Electronics and Communication Engineering,
Shanmuganathan Engineering College, Pudukkottai, Tamilnadu, India
anithasn15@gmail.com,punikalyani@gmail.com

Abstract— Data security is vital in PDAs and Smartphones. There are lots of mobile applications available in the market. Every application has its own declassification policies for data access and security. Some application may violate those policies and users don't know that their private information read by them. The proposed system will ensure mobile applications, data access flow to be secured. Static analyzers introduced an expression-matching framework that defines validity of a program with respect to the expressions calculated from it. The framework checks all the expressions in a given program can possibly output and checks them against a set of expressions which are allowed to be declassified (the set of declassifiable expressions). It formalized a property that states that the program does not reveal any more information than that specified by such declassifiable expressions. It named this property Policy Controlled Release (PCR).It has also developed a high level implementation of the framework, which uses a form of graphs to calculate whether a program satisfies PCR or not. The PCR property that the framework refers to is undecidable. Expressions that may be computed by the program under analysis are also represented by a form of an expression graph that incorporates representations of variables and I/O channels, and captures the dependencies of output expressions on values obtained from input channels, runtime components to achieve an enforcement mechanism that can be applied to current technologies and application examples. In order to tackle some aspects of information flow enforcement that static analysis does not cover, the implementation with runtime enforcement techniques.

***Index Terms*—Data security, Information Security.**

I. INTRODUCTION

Computer systems are constantly handling sensitive information. As most of such systems are networked and often connected to the Internet, sensitive data is also regularly transmitted between different devices. Smartphones and tablet computers carry private data of users in the form of contact lists, photos, messages and others. Social network websites not only store such information, but also regulate who has access to it. Banking systems are responsible for securing and regulating access to very sensitive financial information of their customers. Wrong handling of sensitive information can cause it to be disclosed to unauthorized parties, corrupted or lost. This can cause major loss for both companies and individuals.

The field of computer security can be divided into 3 main aspects are confidentiality, integrity and availability. Confidentiality is related to ensuring that data is only accessible by entities authorized to do so. Integrity is about preventing that data gets corrupted or modified in unauthorized ways. Finally, availability is about guaranteeing that computer systems and services are available at all times. This thesis tackles the first aspect, confidentiality of sensitive data. The aim of the problem is ensuring that programs do not leak sensitive information to unauthorized entities.

Software is the fundamental decision-making component of a computer system. Every action done with data, including modification, copy, delete and transmission is done by programs. Thus, in order to regulate actions over data, one needs to regulate how computer software operates over such data. To make matters more complicated, it is common for a computer system to have a multitude of different programs, which are in turn also regularly updated. Thus, in order to enforce how sensitive information is handled by a computer system, one needs to regulate what programs do with this information.

Information can have different degrees of confidentiality. In a company, some information might be of public domain, e.g. the company's line of products and services, its address, some general numbers about profits. However, an employee should not be able to access another employee's salary information, while a manager should be able to access this information related to all his/her subordinates. Some information might be even more sensitive details of unannounced research projects or the company's financial situation should be accessible only by some key personnel. With this, there is a need for computer systems to regulate "who" (i.e. programs working on behalf of users) can access which kind of information.

INFORMATION FLOW AND DECLASSIFICATION ANALYSIS

Programs dealing with sensitive data must prevent confidential information from flowing to unauthorized entities. In order to enforce how programs use data, information flow control has become increasingly popular within the scientific community. Information flow control revolves around a classical security property called non-interferenc*e,* which states that the publicly observable behaviour of a program is entirely independent of any secret input values it has received. Several techniques have been proposed to check whether programs satisfy this property, within both static analysis and runtime enforcement.

Consider a program with two inputs are one which is publicly observable labelled low, and therefore not confidential, and another which is secret, labelled high, and whose contents should be disclosed to unauthorized entities. This program has one public output, labelled low. It could also have secret (high) outputs, but these are unnecessary for the sake of this example. Consider that P (l, h) returns the program's public output for when it is executed with the input values l and h, for the low and high inputs, respectively. User says that this program satisfies non-interference if, for any two executions differing only in the value of the high input, the value of the low output does not change.

In other words, for any l, h and h0, user have that P (l, h) = P(l, h0). In this case, user say that the secret input does not interfere with the value of the public output, and thus this program does not perform any unauthorized information flow. Most of the approaches to guarantee information flow can be divided into two main categories are static analysis and runtime enforcement. Static analysis consists of analyzing the program's code in order to predict its behaviour, while runtime enforcement revolves around checking, during runtime, every instruction executed by the program, and taking action should an unauthorized action takes place. These two approaches are complementary some aspects of information flow can only be tackled by static analysis (e.g. implicit flows) while others only by runtime enforcement (e.g. resources with labels only known at runtime). Information flow controller and security policies enforcer for mobile application has three mechanisms, each building upon its predecessor, from theory to practice as following

1. A theoretical framework that defines a policy model and the notion of program validity with respect to a policy.
2. A high level implementation that defines a concrete policy language and a tractable validation procedure for checking program validity against such policies.
3. A practical extension of the implementation, that defines a framework which combines the previous mechanism with a runtime component implemented on a established technology, supporting more expressive policies across multiple systems, but keeping runtime overhead very low.

II. RELATED WORK

Various schemes have been proposed for contextual property detection. In [2], the Gradual Release (GR) property [10] formalized the information revealed by declassification policies by stating that the observer's knowledge increases only at declassification points. This property was extended by the Conditioned Gradual Release (CGR) property [10], which took important steps in decoupling policy from code. It requires that the low-security observer of program behavior cannot detect differences between runs whose inputs yield the same values for declassifiable expressions. Finally, a more complete separation between code and policy was achieved by the Policy Controlled Release (PCR) property [10] which, based on CGR,is able to completely remove security types from the program.The PCR approach is the basis of our static analyzer.Runtime enforcement mechanisms [8] monitor accesses a program does during execution, enforcing access control policies. These mechanisms are often useful for enforcing access control, but not information flow, since the latter requires knowledge of nonexecuted code, in order todetect implicit flows. In [8] authors propose a theory for runtime enforcement, modelling runtime mechanisms that can transform results, and also an analysis of the policies that such model can enforce. Their abstract model is simple and expressive, and our runtime enforcement step can be fit in the model in a straightforward manner. The model, however, makes explicit one of the limitations of runtime enforcement,as it only considers actions performed by the application at runtime, it is unaware of implicit flows of information caused by actions that were *not* performed. A recent study on policies enforceable by runtime monitoring is presented in [5]. The same authors present a framework for composing expressive runtime policies in [3]. However, policies are again based on specific security-sensitive actions performed by the program.In [6], the authors address the issue of the computability constraints of runtime monitoring, giving a characterization of those security policies enforceable by program rewriting. In [8] the authors propose a purely dynamic information flow analysis approach that handles implicit flows. However, this is achieved by disallowing, on the language semantics, dynamic label updates within high conditionals, an unnecessary limitation in our approach. In [9], authors study language support for runtime principals that specify runtime authority in downgrading mechanisms such as declassification. we establish the basic property of noninterference for programs written in such language, while an example of a security sublanguage for enforcing information-flow policies was proposed in [4].Finally, in [2] the authors present a semantic framework for expressing security policies for declassification and endorsement in a language-based setting. The proposed framework specifies how attacker controlled code affects program execution and what the attacker is able to learn from observable effects of the code.A information controller and security policies approach had been proposed in [1], although authors proposed the combination of inline reference monitors with static type systems.

III .MOTIVATING EXAMPLES

We present three examples that will be used throughout the paper. The examples are all within the context of mobile devices, and present problems which current popular mobile platforms (e.g., Android, Apple iOS) cannot handle. Indeed,these problems cannot be handled by either static or runtime enforcement approaches, emphasizing the necessity for a combined approach.

Example 1 (Classification): Consider a policy that allows applications to read the contents of the phone's contact list, but not send it to low level channels (e.g., an arbitrary Internet connection).However, assume the user is allowed to mark as "trusted" certain output locations, such as a network connection, an SMS or an e-mail address. Thus, information derived from the contact list can only be sent to trusted output channels. In this scenario,the static analyzer is needed todetect the flows of information within a program, while the runtime enforcer is needed to check the dynamic security label of the output channel. Algorithm 1 presents an example. In the following example algorithms we use underlined text to indicate input and output operations.

Algorithm 4.1: Classification application

```
1 clist := getContactList();
2 counter := 0;
3 while hasNext(clist) do
4 contact := next(clist);
5 age := getAge(contact);
6 if age > 45 then counter := counter + 1;
7 text := "I have " + counter + " contacts over 45.";
8 addr := readFromInput();
9 sendSMS(addr, text);
```

Example 2 (Declassification): Consider a policy for location-based services. The policy states that a user's location is private in general and cannot be output. However, there are two allowed declassifications: (1) the time zone of a location, and (2) the result of a function that compares whether two locations are near to each other. In this scenario, an application can transmit its location to a different device using a secure connection.In particular, the application transmits data along with its corresponding security label to the other device (assuming that the underlying system platform supports this). Here, the static analyzer not only detects flows of information, but also points of the program that match the expressions allowed by the declassification policy. Again, the runtime enforcer checks for dynamic labels.

Algorithm 4.2: Declassification application

```
1 secureConn := secConnect("otherhost.somewhere.com");
2 myLoc := getLocation();
3 myTz := timezone(myLoc);
4 otherTz := recv(secureConn);
5 if myTz = otherTz then
6 send("ACK", secureConn);
7 otherLoc := recv(secureConn);
8 near := isNear(myLoc, otherLoc);
9 if near then print("Host is nearby!");
```

Example 3 (Iterative declassification): Now, consider a corporate application (Algorithm 3) in which a device accesses the records of several products, and it outputs the average of some property of the products (e.g., price, nutritional facts, cost, etc.).According to a declassification policy, the program can only output the average of a property for a given number of products (and not their single values). The static analyzer detects that the program conforms with the declassification policy, but the condition of the minimum amount of values the average has to contain is only checked during runtime.

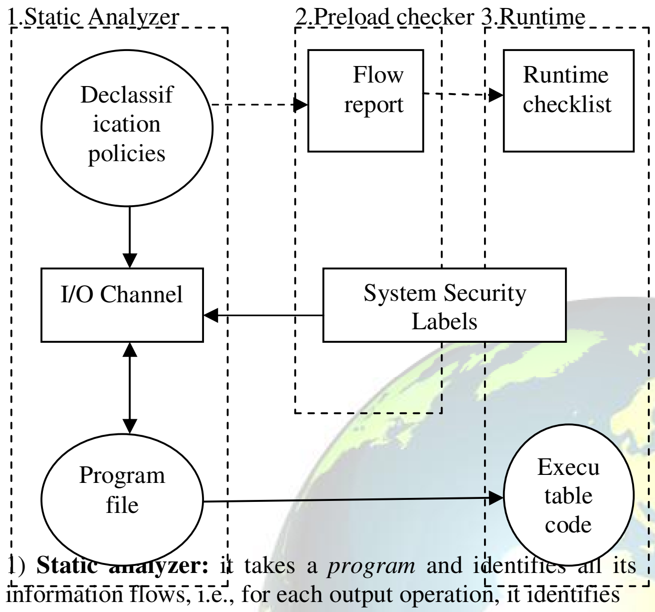
Algorithm 4.3: Iterative declassification application

```
1 sum := 0;
2 num := 0;
3 db := openDBConnection();
4 while !exitSignal do
5 rec := fetch(db);
6 prop := getProperty(rec);
7 sum := sum + prop;
8 num := num + 1;
9 avg := sum ÷ num;
10 output(avg);
```

III. APPROACH

SecurityPolicies Enforcer

Our approach consists of a hybrid static-runtime mechanism organized in three steps: static program analyzer, preload checker, and runtime enforcer. In practice, the first two steps perform the most expensive part of the analysis, leaving the runtime enforcer to perform a few very precise (and thus efficient) checks. Fig. 1 shows how the three steps interact with each other, while in the following we give an overview of their role.



1) **Static analyzer:** it takes a *program* and identifies all its information flows, i.e., for each output operation, it identifies which input operations its value can potentially depend on (including implicit flows). Additionally, it takes a set of *declassification policies* and identifies which variables of the program hold expressions on inputs allowed by the policies. Thus, it downgrades the security level of those variables and of the corresponding flows of information. The information flows, combined with the matched declassifications,are included in a *flow report* of the program.

2) **Preload checker:** before the program is run, the checker takes the flow report from the previous step and checks the *security labels* of the system in which the program is about to run. The information flows with static labels are then validated at this step (i.e., `high` cannot flow to `low`).Flows containing I/O channels with dynamic security labels can only be checked at runtime, and thus are marked for checking in a *runtime checklist*. Also, declassifications from the previous step might have constraints associated with them, some of which may only be checked at runtime.

3) **Runtime enforcer:** the lightweight enforcer verifies that the conditions of the runtime checklist are satisfied at certain points of execution. The conditions may consist of checks of security labels of channels as they are accessed,and also of counting the number of times some loops in the program run. In order to reduce runtime overhead, the calls to the enforcer are injected in the application bytecode,prior to the program's execution, on the specific program.

Contribution. We present a information flow and security policies mechanism for mobile application approach for information flow policies, including declassification,which satisfies the points above. Our mechanism has 3 stages: (1) a static analyzer that takes a program source and a set of declassification policies and detects all flows of information between input and output channels in the program,as well as detecting points where declassification can happen (generating constraints that have to be checked at runtime);(2) a preload checker which, before loading the program for execution, checks the security labels of I/O operations specific to the target system against the information obtained in the previous step; and (3) a runtime enforcer that checks labels which are only known at runtime, as well as runtime constraints for the declassification policies. Calls to the enforcer are injected in the application's code, prior to its execution, on the specific points where checks are needed, thus further reducing the overhead of the enforcer. To the best of our knowledge, our is the first proposal of such approach.We present three motivating examples, all within the context of a mobile device, and show that our hybrid static-runtime enforcement suffices to:

• support more realistic policies than present approaches—as policies may need both static (implicit flows, declassification) and runtime (dynamic labels,execution constraints) knowledge
.• reduce runtime overhead—as most of the analysis computation is done statically, and the static analyzer is system independent;

With this, we fill the gap left by existing approaches and demonstrate that information flow and declassification analysis can be performed on real-world scenarios (i.e., legacy, untrusted and mobile code). We combine the static analyzer with a runtime component in a nonstandard way, in the sense that we also include an intermediate step between both stages and perform runtime enforcement via a code injection done only in thempoints of the code where enforcement is necessary. This is opposed to the standard definition of runtime enforcement[5] ,in which everyı program instruction needs to be monitored.We show how this nonstandard approach allows us to have a system independent static component, while also having an extremely lightweight runtime component. In particular, we argue that our approach has the following advantages: (1) it does not require specially annotated program code, (2) handles information-flow at the level of program variables, (3) supports declassification policies which are decoupled from the code, (4) performs a system-independent static analysis, due to the presence of system-specific labeling mechanism that is decoupled from program and policy, (5) supports dynamic (runtime) security labels,(6) handles runtime declassification constraints, and (7) its runtime component is lightweight enough to be implemented on mobile devices.

Overhead. We have implemented our runtime enforcer in java, and measured both its processing and memory overhead, running with applications on an Android device. First, we discuss the theoretical limits for this overhead, and then we proceed to show our experimental results. For the memoryoverhead, the enforcer keeps two buffers, and , which map a program point to an integer and a label,respectively. These buffers can be implemented either with standard arrays or hash tables. Note that entries on each of the two buffers point to different types of commands: entries in point to looping and entries in to input commands. So, a worst-case scenario happens on a program made entirely by loops and inputs, all loops being referenced by policies, all inputs being dynamic, and a single output in the end, with all inputs flowing to it. In this case, for a program with commands, exact entries are made on the buffers, each using one memory word (32 or 64-bit).

Note that, in practice:

(1) the average case tends to use considerable less emory,e.g., in our 3 examples, the ratios of (number entries/number commands) were 0/9, 1/9 and 1/10, respectively; and

(2) programs tend to use much more memory for their data than for their code, meaning that the bound of entries in the buffers is usually low.As for the processing overhead, note that each injected code piece is a simple call to one of the enforcer's methods. These methods, in turn, are implemented with the execution and verification of a simple statement, with no loops. Thus, it is clear that the enforcer methods have, by themselves, constant complexity,and that the enforcer does not change the complexity of the monitored program. Once again, the number of checks added to the program is bounded by the number of commands.But most practical cases do not reach the bound , since only operations on dynamic I/O channels and declassification constraints generate checks. In our 3 examples, the ratios of (number checks/number commands) were 1/9, 3/9 and 2/10, respectively.It should be noted that, in the classical definition of a runtime execution monitor [5], the runtime enforcer monitors *every* command of the program. Our enforcer, though, does not necessarily need to monitor every instruction, since the task of identifying instructions that need monitoring is performed by the previous stages of our hybrid approach.We have implemented Android versions of the three examples of this paper, plus a number of benchmarking programs meant to stress the runtime enforcer performance. Unfortunately,there are only a few proposals for hybrid approaches in literature, and they all differ not only in how they are measured,but also on their specific goals. Thus, there is not yet a "standard benchmark" for hybrid static-runtime information flow and declassification analysis, making a direct comparison of performance with other approaches not possible at this moment. Our experiments have the purpose of showing that the overhead of our runtime component is negligible for most practical scenarios.

We recall that the aim of our hybrid-approach is to minimize the runtime checks only to methods that are relevant and have not been covered by the static analysis phase. Typically these represent a fraction of all methods invoked by an application at runtime. Thus, running the experiments using legacy applications would have shown a smaller overhead than the small, security intensive benchmark programs we use here. To stress and focus the performance penalties due to our runtime check, we decided then to implement our own applications representing the three motivating examples shown through the paper. Furthermore, in order to also consider worst case scenarios, uncommon in real-world applications, we implemented the ad-hoc applications *FileCopy*, *FileEncrypt*, *InfGather* and *Statistics*,with the specific purpose of stressing the enforcer and computing the overhead in these extreme cases. Results show that even in these extreme cases, our hybrid technique has limited overhead compared to dynamic analyses that intercept and analyze all methods that are invoked at run time.

Each of the benchmarkswe implemented has a different "profile"for accessing I/O. *FileCopy* performs a copy between files,reading blocks of 1 KB at a time. However, each block has a `data` security label. Thus, the runtime enforcer has to set the label of each write with the label from the previous read. This is an example of a program with extreme I/O access, all of which checked by the runtime enforcer. *FileEncrypt* is the same as the previous, but each block is encrypted before being written.With this, the program incurs a considerable processing time between I/O accesses. *InfGather* and *Statistics* are similar programs, which access inputs from 10 different sources, and then perform a single output, whose value depends on all previous inputs. In the former, all input channels have `runtime` security labels, which have to be checked during access, and then compared to the output label. In the latter, labels are static,but violate noninterference. However, some statistical calculation is done over the data, and a declassification policy allows such computation. Thus, the runtime enforcer is left to check if the input channels accessed by the program match the ones described by the policy, and also count the number of input accesses made by the main loop. Finally, *Loops* is a program made by several loops, all of which are small in size and have their number of iterations counted by the enforcer, presenting an extreme example of almost every instruction being checked.For completeness, in Table V we report statistics about the size of applications used in our evaluation.

ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com
International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 3, Special Issue 20, April 2016

TABLE V

STATISTICS FOR BENCHMARK APPLICATIONS

Program	Original code size (bytes)	Code size with enforces (bytes)	Number of methods
Example1	1,902	2,196	6
Example2	3,252	3,662	9
Example3	1,429	1,671	6
FileCopy	782	981	2
FileEncrypt	2,013	2,211	5
InformationGather	1,185	1,499	2
Statistics	1,223	1,632	2
Loops	1,367	2,192	2

IV.CONCLUSION AND FUTURE WORK

Information flow controller and security policies enforcer for mobile application designed to support policies that need both static and runtime information. It works on minimum runtime overhead in smart phones. Additionally, it does not require specially annotated code. Static analysis stage supports declassification policies which are decoupled from the code. In this stage Graph based policy controlled release mechanism is introduced. Preload checker stage verifies the flow report compare with system security labels. Run time enforcer code injection method is used to change unsecure download application to secure application, if it's any unsecure code is identified.We are implementing in android based devices.In future other type of mobile devices we are considered to implement.

REFERENCES

[1] Askarov.A and Myers.A, (2010), "A semantic framework for declassification and endorsement," in Proc. ESOP'10, pp. 64–84.

[2] Askarov.A and Sabelfeld.A, (2009), "Tight enforcement of information-release policiesfor dynamic languages," in Proc. CSF'09.

[3] Banerjee.A, Naumann.D.A, and Rosenberg.S, (2008),"Expressive declassification policies and modular static enforcement," in Proc. SP'08, pp. 339–353.

[4] Chong.S and Myers.A.C, (2008),"End-to-end enforcement of erasure and declassification," in Proc. CSF'08, pp. 98–111.

[5] Conti.E, Fernandes.E, Crispo.B, and Zhauniarovich.Y, "CRePE,(Oct.2012)," A system for enforcing fine-grained context-related policies on Android,"IEEE Trans. Inf. Forensics Security, vol. 7, no. 5, pp. 1426–1438.

[6] Davi.L, Dmitrienko.A, Egele.M, Fischer.T, Holz.T, Hund.R, Nürnberger.S, and Sadeghi.A.R,(2011), "Poster: Control-flow integrity for smartphones," in Proc. CCS'*11*, pp. 749–752.

[7] Erlingsson.U and Schneider.F.B, (1999), "SASI enforcement of security policies:A retrospective," in Proc. NSPW'99, pp. 87–95.

[8] Ligatti.J and Reddy.S, (2010), "A theory of runtime enforcement, with results", in Proc. ESORICS'10, pp. 87–100.

[9] Rocha.B.P.S, Bandhakavi.S, Den Hartog.J, Winsborough.W.H., and Etalle.S, (2010), "Towards static flow-based declassification for legacy and untrusted programs," in Proc. SP'10, 2010, pp. 93–108.

[10] Tse.S and Zdancewic.S, (Nov.2007), "Run-time principals in information-flow type systems," ACM Trans. Program. Lang. Syst., vol. 30, no. 1.