Hybrid Static-Runtime Information Flow and Declassification Enforcement

Bruno P. S. Rocha, Mauro Conti, Sandro Etalle, and Bruno Crispo

***Abstract*—There are different paradigms for enforcing information flow and declassification policies. These approaches can be divided into static analyzers and runtime enforcers. Each class has its own strengths and weaknesses, each being able to enforce a different set of policies. In this paper, we introduce a hybrid static-runtime enforcement mechanism that works on unannotated program code and supports information-flow control, as well as declassification policies. Our approach manages to enforce realistic policies, as shown by our three running examples, all within the context of a mobile device application, which cannot be handled separately by static or runtime approaches, and are also not covered by current access control models of mobile platforms such as Android or iOS. We also show that including an intermediate step (called preload check) makes both the static analysis system independent (in terms of security labels) and the runtime enforcer lightweight. Finally, we implement our runtime enforcer and run experiments that show that its overhead is so low that the approach can be rolled out on current mobile systems.**

***Index Terms*—Data security, information security.**

I. Introduction

COMPUTER systems often run applications that can access protected data. This is the case not only for standard systems but also for portable devices like PDAs and smartphones (such as iPhone and Android platforms). On these devices, applications usually have access either granted or denied for each individual resource requested to the system.

However, such access control mechanisms are not enough to control how data is used. Since applications often have to deal with both secret (e.g., pincode) and public (e.g., phone number) data, in these situations it is crucial that secret (*high*) information is not leaked to public (*low*) output channels (noninterference [23]). Noninterference is in turn usually too strict to be practical (even when not considering side-channels): many real programs eventually need to downgrade the security level of some information. For example, consider a company policy requiring that the salary of an employee should be considered as confidential data: any data obtained from a function that takes this salary as an input will be considered confidential. As a result, information such as the average of salaries, overall human resource expenses, or company balance cannot be made public. For this reason, *declassification policies* [52] are needed, identifying conditions under which high labeled data can flow to low labeled channels.

Several *static* approaches to check control-flow integrity and declassification have been proposed. These are based on analyzing the program source code before runtime, via type checking [60], [40], [43] or dataflow analysis [19], [2], [1], [9]. Programs that fail to satisfy the control-flow integrity or declassification policies are then discarded before execution takes place. A slightly different approach is presented in [25], where the authors present an extension to the .NET CIL that makes it possible to automatically verify an in-lined reference monitor (IRM) using a simple static type-checker, hence eliminating the need to trust the producer of the IRM. This is possible via typing annotations that track an abstract representation of a program's execution history.

This type of analysis has the drawback of not being suitable to enforce several desired policies. For instance, in the previous example of the average salary, it might be desirable to have a constraint on the minimum number of salaries considered in the calculated average, which is not known until runtime, and thus enforcement needs runtime information. Also, security labels can vary between individual systems, forcing static analysis to be done directly on the target system, an undesirable restriction for programs aimed at portable devices. Finally, some data may have dynamic security labels, which might change during execution, e.g., a file access by the program, in which the name of the file is only known during *runtime*.

To solve these issues, runtime approaches have been proposed [30], [29], [36], [16], [47], [48]. These approaches are however often computationally expensive, incurring a nonnegligible overhead on the applications. Also, they fail to achieve some features of static analysis, such as detecting implicit flows and declassification. Another approach is (automatic) program-rewriting [24], [35], [61], which satisfies the security policy during runtime, by rewriting instructions that violate it. In particular, [24] considers rewriting bytecode, [61] works on binary code, and [35] proposes both source and binary rewriting. A rewriting approach, by itself, shares some of the limitations of the previous ones. For instance, it is unable to detect implicit flows of information, only captured by a static analyzer that considers the whole code, rather than just the executed instructions. Also, rewriting faces the additional problem of possibly

Manuscript received September 02, 2012; revised February 23, 2013; accepted May 29, 2013. Date of publication June 12, 2013; date of current version July 09, 2013. This work was supported in part by the S-MOBILE project (Grant VIT.7627) funded by the Dutch STW-Sentinel, and the TENACE PRIN Project (Grant 20103P34XC) funded by the Italian MIUR. The work of M. Conti was supported by an E.U. Marie Curie Fellowship for the project PRISM-CODE (Grant PCIG11-GA-2012-321980). The associate editor coordinating the review of this manuscript and approving it for publication was Prof. T. Charles Clancy.

B. P. S. Rocha is with the Eindhoven University of Technology, Eindhoven, 5612 AZ, The Netherlands (e-mail: brocha@palantir.com).

M. Conti is with University of Padua, Padua, 35131, Italy (e-mail: conti@math.unipd.it).

S. Etalle is with University of Twente, Enschede, 7500AE, The Netherlands (e-mail: sandro.etalle@utwente.nl).

B. Crispo is with University of Trento, Povo (TN), I-38123, Italy (e-mail: crispo@disi.unitn.it).

Digital Object Identifier 10.1109/TIFS.2013.2267798

changing the semantics of the monitored program, a side-effect which is not always tolerable.

When it comes to handling real world programs all approaches above share the following limitations:

- Type-based approaches that support both information flow control and declassification require annotations of declassification operations. Thus, programs must be written by programmers aware of both the analysis technology and the declassification policies that will be applied to it. Dataflow approaches (which analyze unannotated code) and runtime enforcers are not currently capable of handling declassification in an automatic fashion.
- While recent work explicitly considers *mobile* code [3], [27] (which is platform independent), we believe this remains an important and intricate problem, in particular in terms of reducing the computational overhead on resource-constrained devices such as smartphones. For instance, recent proposals [3], [27] do not consider the specific constraints of mobile platforms, and also provide validation of their proposed solutions via experiments run on classic desktop-computer platforms.

These limitations make information flow and declassification analysis impractical for real-world programs. In order to make such analysis possible, a mechanism needs to be able to handle legacy, untrusted and mobile code. Recent work by [46] has made an important advance in this goal by proposing a static analyzer that works on unannotated code, supporting declassification policies which are decoupled from it. That work, however, has some limitations of its own: the static nature of the approach limits the kinds of policies that can be enforced and also makes it system dependent, limiting the applicability to mobile devices.

In order to be able to analyze legacy, untrusted and mobile code, the following goals need to be satisfied: i) providing information flow control, ii) with support to declassification policies, iii) on unannotated programs, iv) with support to runtime security labels, v) with a system independent static analysis phase, and vi) with little runtime overhead. Some hybrid approaches have been proposed in the literature [49], [7], [14], [62], [41], but none of them achieve all our goals.

Contribution. We present a hybrid static-runtime enforcement 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 the points of the code where enforcement is necessary. This is opposed to the standard definition of runtime enforcement [53], in which every[1] 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.

This paper is guided by the three examples and the approach is presented in an implementation-oriented fashion. As for the static analyzer needed in our approach, we explain how to extend PCR analysis [46] so that it can be integrated with the other steps of our approach. As for the preload checker, we show how the runtime enforcer checks are injected in the application's code. Finally, we implement our final step, the runtime enforcer, which is the most significant component in terms of overhead. We run our runtime enforcer against benchmark programs on an Android device, in order to determine its overhead. We show that the resulting overhead is almost imperceptible. The focus of this paper is to propose a solution that combines the strengths of different approaches (static analysis, code injection, and runtime enforcement). We present our approach with a focus on a practical description, rather than a formal one. However, we do believe that there is still the need for a formal proof for our complete system, which we leave as future work. We underline that while each single component has been separately studied (e.g., the proof for our static analysis component is already given in

[1]In the standard definition found in [53], each step the monitored target is about to take generates an input symbol, which is sent for evaluation to a security automaton simulation: since the program counter is a state component and it changes each time a machine-language instruction is executed or an interrupt occurs, the enforcement mechanism must be involved in executing each target instruction. Possible optimizations on when to issue the symbol are also envisaged in [53]: e.g., in case of enforcement mechanism for access control policies on files, the production of input symbols can coincide with occurrences of file access operations.

[46]), proving the soundness of the combined techniques is a separate contribution. Also, typically the 3 different approaches are each proven by a different proving mechanism, further outlining the specific challenges of a formal proof for the whole hybrid approach.

We underline that our approach of splitting the problem in three steps and coordinating these activities does not make the problem tractable or decidable—undecidability is intrinsically part of the problem addressed in this paper [31], [45]. However, as shown through the examples, we can solve a number of instances of the general information-flow and declassification problem which cannot be treated by only static analysis, runtime enforcement or program rewriting, separately. We acknowledge that the general problem is undecidable and that our approach does not present solutions to every instance of the problem. And by presenting the approach in a practical manner we (1) show the high practical applicability of the instances we solve and (2) demonstrate that the combination of the 3 analysis approaches is not trivial.

The remainder of the paper is organized as follows. In Section II, we review the state of the art of information-flow and declassification analysis. Section III presents our practical motivating examples (which are also referred throughout the rest of the paper). Section IV describes some preliminary assumptions. Section V describes declassification policies and presents an overview of our approach. Then, we present each of the three components in our solution. In particular, Section VI describes how we adapt an existing static analyzer to be used as the first component. The following sections describe more in detail the preload checker component (Section VII), and the runtime enforcer component (Section VIII). Finally, in Section IX we draw the conclusions of our work.

II. Related Work

Most of the static analysis approaches in literature revolve around annotating program variables with security types [43], [60], [11]. Jif [15] is one of the most advanced programming languages designed to enforce fine-grained declassification policies in the program. However, if the programs and policy are not carefully designed, both get interdependent, with changes to one incurring changes to the other [28]. The use of type-based languages incur that the programmer not only knows a more intricate language, but is also aware of the policy. This assumption is often not desirable, such as in the mobile device scenario we use in our examples. Even though Jif supports polymorphic label-types in the code, enforcement of labels that can change during runtime is not possible, due to the nature of static analysis. Dataflow [2], [1], [9], [19] and taint [39], [58], [20] analysis perform static analysis on unannotated code, but fail to support declassification policies. Also, implicit flows of information are often harder to deal with. In [55] authors present a λ-calculus based language for dynamic information flow tracking, that accepts more programs than type-based systems, at the cost of greater overhead. Their approach tracks information flow in multiple dimensions (i.e., it reasons over, e.g., the confidentiality of an integrity label), a goal out of the scope of this paper. Even though variables have no static security labels, declassification is done explicitly in the code, by the programmer.

Support to declassification often needs some help from the programmer. Type-based enforcement approaches [50], [6] keep track of variables that hold declassifiable expressions via extensions on the type system. The Gradual Release (GR) property [5] 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 [12], 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 [46] 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 [36], [16], [29] 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 to detect implicit flows. In [38] 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 [37]. The same authors present a framework for composing expressive runtime policies in [13]. However, policies are again based on specific security-sensitive actions performed by the program. In [26], 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 [59], authors study language support for runtime principals that specify runtime authority in downgrading mechanisms such as declassification. They 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 [34] 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 hybrid approach had been proposed in [54], although authors proposed the combination of inline reference monitors with static type systems. Our approach precludes the need of a type system—even though a type system might be necessary to undertake a formal proof for our solution (which is left as future

work), but not in the static analyzer itself. Concrete proposals of hybrid mechanisms are scarce, although have become increasingly popular [32], [33], [56], [42]. In [63] the authors integrate static analysis and runtime tracking to establish an approach to generate a sensitive data propagation graph aimed at incurring minimum time overhead on systems. All the cited approaches, however, do not support either runtime security labels or declassification policies. In [44] the authors use static analysis to detect which parts of the program satisfy a policy, and use a runtime enforcer to guarantee that unsafe parts are not executed. Thus, they do not enforce policies that need runtime information: the runtime enforcer serves only to select the parts of the code that may be executed.

In [51] the authors show that, by blocking execution of unsafe instructions, a dynamic monitor can guarantee termination-insensitive noninterference, for a flow-insensitive analysis. Then, in [49], the same authors prove impossibility of a sound purely dynamic information-flow monitor that accepts programs certified by a classical flow-sensitive static analysis. The authors demonstrate the need for hybrid mechanisms in flow-sensitive analysis, and present a general framework for such mechanisms. In both papers, however, authors do not consider either declassifications or dynamic labels. Hybrid mechanisms that support declassification have been proposed in [7], [14] however these approaches do not share the expressiveness of PCR analysis, being unable to declassify expressions of unbounded size (such as the average salary), do not share our goal of separating policy from program, and work on security typed languages, requiring the programmer to identify points where declassification occur.

A recent implementation-oriented approach is Resin [62], a language runtime that implements data-flow assertions. It is a fully runtime approach, incurring a nonnegligible overhead (33% CPU overhead for their measured application). Besides, it does not share a number of our goals: it allows the programmer to specify application-level data flow assertions, as opposed to our goal of analyzing untrusted programs, and it does not focus on information flow control or declassification policies. Similar observations hold for TaintDroid [20] which performs dynamic taint analysis for the Android system (which is the same system we considered for our implementation). TaintDroid assumes that third-party applications are not trusted and monitors how the applications propagate sensitive data. However, differently from our work, TaintDroid is not capable of enforcing declassification, or fine-grained policies (to let only specific tagged data to flow to application or to network connections). Finally, another recent implementation of control-flow integrity, with smartphone architecture in mind, has been proposed in [18]: again, also this solution only consider runtime enforcement and does not allow declassification policies.

III. Motivating Examples

In this section, 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 to detect 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 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. See Algorithm 2 where *isNear* only works with arguments from a location input (such as a GPS).

Algorithm 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 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);
```

IV. Preliminary Assumptions

In this section we describe the assumptions made by our work. First, we discuss two important concepts used in the examples, and throughout the paper.

Security labels. Each input/output channel accessed by the programs has a security label associated with it. Security labels are not annotated directly on the program code because labels are system independent (hence not known at coding time) and our work deals with unannotated code. The labels form a lattice [19] in which they are ordered, from least restrictive (lower) to the most restrictive (higher). For simplicity, in this paper the only static labels are `low` and `high`. For the standard information flow policy, information can only flow from low to high in the lattice.

Declassification policies. Exceptions to the standard information flow policy are defined by declassification policies. Declassification policies are (allowed) exceptions to the information flow policy in force. A declassification policy may specify that—under some specific conditions—some `high` data may be relabeled as `low`. The "specific conditions" depend on the functions that have been applied to the data, describing which expressions on the input values may be declassified. Thus, if a variable on the program holds the expression described by the policy, it is allowed to be sent to a `low` output channel.

Now we describe the assumptions made on the underlying system, which is part of the *Trusted Computing Base*. The considered programming language is assumed to have a well-defined set of I/O statements, which can be identified by the static analyzer. These I/O statements are "safe", in the sense that their behavior is always the expected one. Also, functions referred by declassification policies (such as $timezone$, in Example 2) are also safe, meaning that they can not be abused or inverted in order to obtain the original value of its arguments. Policies using unsafe functions are considered malformed policies [52], and measuring the safety of a declassification policy is out of the scope of this paper. We remind that, while no inliner for multithreaded Java can be both secure and transparent for all the possible policies, this is possible for a broad class of policies [17], to which we restrict our study.

We consider an underlying system that includes a security labeling system, and provides an API for handling the labels. We leave the implementation of this API unspecified, since this paper focuses on the enforcement of policies by programs, rather than on the specification of a labeling system [40], [10], [57]. Thus, the API is composed of commands to get and set (when the program is permitted to) security labels on data blocks (e.g., files, network packets) and also to determine labels of I/O channels (e.g., a network or DB connection, a phone's camera or SMS manager). Also, we consider operators $\sqsubset$, $\sqsubseteq$, $\sqsupset$ and $\sqsupseteq$ for comparison of labels. In this paper we consider I/O channels and data blocks can have labels `hi` and `low`, besides the special labels:`runtime` for an I/O channel whose label can only be known at runtime, and `data` for channels in which each individual packet of data has a label attached to it.

V. Declassification Policies

We define a simple language for specifying declassification policies. Policies in this language are easily converted to the graph format used by the static analyzer. For Example 2, we have the following declassification policy:

```
define
  alpha = input:getLocation()
  beta = input:any
allow
  output:low := timezone (alpha)
  output:low := isNear (alpha, beta)
```

The keywords `input` and `output` are used to represent I/O channels. The `define` block names the input channels cited by the policy, and the `allow` block defines operations that should be permitted. Here, two expressions on the inputs are allowed to be sent to `low` output channels. Next, we present the policy for Example 3:

```
define
  alpha = fetch(openDBConnection())
  c >= 25
  x = 0
allow
  for i = 1 to c do x := x + alpha
  output : low := x
```

In this policy, we use local variables to hold intermediate values. The `define` block initializes these variables. Note that c can be any number greater or equal than 25. The allowed expression describes the sum of at least 25 values from α. Thus, we have the policy format defined by the grammar:

$$
\begin{aligned}
P &::= \texttt{define}\ D\ \texttt{allow}\ A \\
D &::= IO | v\ op\ n | D_1 D_2 \\
IO &::= \alpha = IO's | \alpha = IO'k \\
IO' &::= \texttt{input}: | \texttt{output}: \\
A &::= C | O := v | A_1 A_2 \\
O &::= \alpha | \texttt{output}: k \\
C &::= s | \text{if } e \text{ then } C_1 \text{ else } C_2 | \text{while } e \text{ do } C \\
&\quad | C_1; C_2
\end{aligned}
$$

Where v are variable names $(x, y), op$ are standard logic and arithmetic operators $(+, -), n$ are numeric constants, α are names for I/O channels (`alpha`, `beta`), s are one-line program statements (assignments and function calls), k are keywords that match multiple channels (`any`, `low`), and e are expressions, defined the standard way. Note that rule C defines a program, the other rules being specific to the policies.

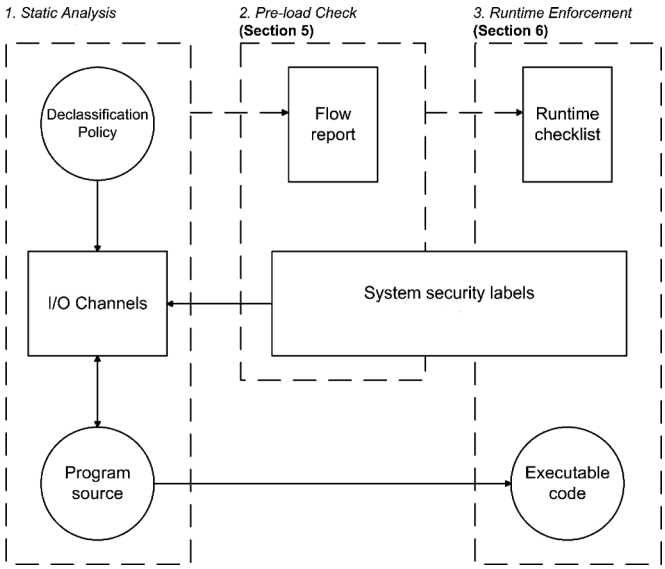


Fig. 1. Overview of the three-step enforcement.

A. Hybrid 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 points that need checking.

VI. The Static Analyzer

Our static analyzer starts from the graph-based Policy Controlled Release (PCR) mechanism introduced by [46]. This mechanism is able to static analyze code in a language with assignments, loops and conditionals, as well as with well-defined I/O commands, like the one we use in our examples. Also, it supports declassification policies which are decoupled from the code, and automatically detects points in the code where the value held by a given variable satisfies a declassification policy (e.g., a variable that holds an average of values from a given input). The graph-based PCR analysis is *flow sensitive*, in the sense that variables do not have fixed security labels associated to them. Also, it is *termination insensitive*, as it does not handle leaks of information caused by observing the termination behavior of the program.

Graph-based PCR analysis is designed by its authors to produce a *yes/no* result. That is, it takes a program and a policy as input and returns whether the former satisfies the latter or not. However, in order to combine it with our other steps of enforcement we need to modify this behavior. We need the analyzer to return a report with all the flows of information found in the program, plus points where declassification happens. We call such report a *flow report*.

A *program point* is used to identify an input (or output) operation in the program, and may be referenced by both the source and the compiled code. For each I/O operation detected by the static analyzer, a wrapper is generated around that operation, ensuring that the same program point used in the source will be recognized in the compiled code. We use Greek letters to identify I/O channels and write θ_i to denote the *I/O operation* on channel θ at program point i. On a high-level source code, a program point can be seen as a line number.

The flow report contains all flows of information in the program, one for each output operation in the program code, including declassification matchings. A flow is basically a relation between an output operation and a set of input operations whose values can influence it. Consider again the program of Example 1: there are two input operations, at lines 4 and 8, and one output at line 9. Here, we name these operations α_4, β_8 and γ_9, respectively. Since in this program the values from both inputs flow to the output, its flow report contain a single information flow, denoted: $\{\alpha_4, \beta_8\} \rightsquigarrow \gamma_9$. This way, flows have the format $I \rightsquigarrow \gamma_p$, where I is a set of input operations, and γ_p is an output operation.

Besides flows from inputs to outputs, the flow report must also contain points where declassification happen. Consider Example 2: here, we denote the input on line 2 as α_2, the two inputs on the same channel made in lines 4 and 7 as β_4 and β_7, and the outputs on lines 6 and 9 as δ_6 and γ_9, respectively. Thus, we could intuitively define two flows of information in this program as $\{\alpha_2, \beta_4\} \rightsquigarrow \delta_6$ and $\{\alpha_2, \beta_4, \beta_7\} \rightsquigarrow \gamma_9$ (note that some flows are implicit, e.g., β_4 to γ_9). However, some variables of this program hold values described by its declassification policy. Recalling the example, we know that variables *myTz* and *near* hold values which are allowed to be declassified. We know that *myTz* holds a derivation of α_2 and *near* a derivation of both α_2 and β_7. Thus, we write the flows as $\{\{\alpha_2\} \mapsto^{X_1} \texttt{low}, \beta_4\} \rightsquigarrow \delta_6$ and $\{\{\alpha_2\} \mapsto^{X_1} \texttt{low}, \beta_4, \{\alpha_2, \beta_7\} \mapsto^{X_2} \texttt{low}\} \rightsquigarrow \gamma_9$. Here, the

term $\{\alpha_2\} \mapsto^{X_1} \texttt{low}$ denotes that the flow from α_2 is declassified to a `low` security label. Since our goal is to have the static analyzer only report flows and declassifications within the program in a system-independent fashion, each declassification is associated to a set of constraints that need to be satisfied in the target system. These constraints are related to linking the input channels described by the policies with the ones accessed by the program. Thus, we have that $X_1 = \{(\alpha_2, getLocation())\}$ and $X_2 = \{(\alpha_2, getLocation()), (\beta_7, \texttt{any})\}$, meaning that, for the declassification to be valid, input operation α_2 must access input channel denoted by command $getLocation()$ and β_7 may match any value. In other words, the expression $\{\alpha_2\} \mapsto^{X_1} \texttt{low}$ will eventually be translated to `low` if every constraint in X_1 is satisfied, and to α_2 otherwise.

Example 3 outlines the other type of declassification constraint we use. Here, the only flow in the static analyzer report is $\{\{\alpha_5\} \mapsto^{X_1} \texttt{low}\} \rightsquigarrow \gamma_{10}$. We know that the value from input α_5 is declassified according to the declassification policy that describes the average. However, the policy contains a constraint about the number of iterations on the assignment of the variable that computes the average. Thus, we have $X_1 = \{(\alpha_5, fetch(openDBConnection())), (4, it \geq 25)\}$, where the second constraint means that the loop on line 4 must iterate at least 25 times.

Finally, we note that the graph-based PCR static analysis has some limitations, which are subject of further research, as pointed by [46]. One of such limitations is the detection of algebraic equivalence between different expressions on the policy and the program. That is, expressions calculated by the program must be syntactically identical to the ones of the policy, for a matching to occur (e.g., $a + b$ is considered to be different than $b + a$). A research path for this problem consists in building a term rewriting system that uses equivalence rules of the considered operations to put both program and policy graphs in a *canonical form*, before matching. We consider this a separated problem, and leave it as future work.

VII. The Preload Checker

The preload checker is the step responsible for matching the report generated by the static analyzer with the security labels of the specific system of execution. Each information flow from the flow report is checked by verifying the labels of the corresponding I/O channels. This is done by using the system labeling API. Flows containing only I/O channels with static labels are validated at this stage, while flows with dynamic labeled channels generate checks to be performed by the runtime enforcer. Also, the static analyzer's flow report may identify declassification matchings which contain additional constraints to be checked. The preload checker also verifies some of these constraints, and the ones that need runtime information are included in the runtime checklist.

The preload checker first translates the elements of the flow report to their corresponding labels in the target system. Each element in the runtime checklist has the format $(i, check)$, where i is a program point and $check$ a directive for a specific check to be performed. The possible directives are detailed in Table I. The preload checker is defined at the end of this section (Algorithm 4), while we first give an intuition of its behavior via our examples. Notice that declassifications that can only be checked at runtime are considered to be necessary, i.e., failure of the check results in halting the program. This is further discussed in Section IX.

TABLE I
Runtime Enforcer Directives

Name	Directive
`count_iter`	Count number of iterations of current command.
`eval(exp)`	Verify validity of expression `exp`.
`compare_ch(cmd)`	Checks whether the channel accessed by the current input command is the same as the channel relative to `cmd`.
`store_data_label`	Store data label of current input operation. If already existent, stores the most strict one.
`store_ch_label`	Store the label of the input channel accessed by current input operation.
`check_input(pp)`	Check if label relative to input operation at program point `pp` is smaller or equal than that of the current output operation.
`set_data_label(label)`	Set the data label of the current output operation as `label`.
`check_output(label)`	Check if channel label of current output command is larger or equal than `label`.

TABLE II
Preload Checker Output for Example 1. (a) Label Translations; (b) Runtime Checklist

Flow	Labels
$\{\alpha_4, \beta_8\} \rightsquigarrow \gamma_9$	$\{\texttt{hi}, \texttt{low}\} \rightsquigarrow \texttt{runtime}$

(a)

Program point	Statement	Check condition
9	$sendSMS(\ldots)$	`check_output(hi)`

(b)

In Example 1, after the static analyzer does its job, the program is then compiled and the analyzer output is used by the preload checker just before the program is executed. The preloader translates each I/O operation to its corresponding label, as shown in Table IIa. However, notice that γ_9 translates to `runtime`, which means that its label can only be checked at runtime (as it depends on the value of variable $addr$). Based on this table, a checklist for the runtime enforcer is also generated, as shown in Table IIb. In this example, the checklist basically states that the label of the output statement of program point 9 needs to be checked and satisfy the constraint of being at least `hi`.

In Example 2 (Declassification), the preload checker must also check the policy constraints, which are all mappings between input operations on the code and the ones specified by the policy. These mapping can be checked entirely at this step, as shown in Table IIIa. Table IIIb shows the translation of the flows to labels. Notice that, although Algorithm 4 translates declassifications to only the policy label (`low` in all examples) when the check of the constraints does not fail, here we always show all the labels involved in the declassification, for clarity. Recall that `data` stands for a label that is set for each transmission, as opposed to `runtime` (used in the previous example) which means that the whole channel has a single security label, which is known only at runtime. Finally, the runtime checklist is presented in Table IIIc, with 3 items. The first tells the enforcer that the data label of input operation at program point 4 needs to be stored for further usage. Then, the second check treats the first flow: the output channel with automatic label must be labeled according to the inputs it depends on. Thus, the check is for the enforcer to assign a label to the data sent by that output operation, as the maximum label of all the inputs it can leak information on, i.e., the maximum between `low` and the data label

TABLE III
PRELOAD CHECKER OUTPUT FOR EXAMPLE 2. (A) DECLASSIFICATION CONSTRAINTS; (B) LABEL TRANSLATIONS; (C) RUNTIME CHECKLIST

Constraint	Status
X_1:	
(α_2, l)	OK
X_2:	
(α_2, l)	OK
$(\beta_7, \texttt{any})$	OK
where $l = getLocation()$	

(a)

Flow labels
$\{\{\texttt{high}\} \mapsto^{\texttt{OK}} \texttt{low}, \texttt{data}\} \rightsquigarrow \texttt{data}$
$\{\{\texttt{high}\} \mapsto^{\texttt{OK}} \texttt{low}, \texttt{data}, \{\texttt{high}, \texttt{data}\} \mapsto^{\texttt{OK}} \texttt{low}\} \rightsquigarrow \texttt{low}$

(b)

Program point	Statement	Check condition
4	$recv(\ldots)$	`store_data_label`
6	$send(\ldots)$	`set_data_label(max(in_label(4),low))`
9	$print(\ldots)$	`check_input(4)`

(c)

of input at program point 4. Finally, the third check deals with the second flow: for output operation of program 9 to be safe, the label relative to input operation at program point 4, stored earlier, must be at most as strict as the label of the output command of program point 9 (which is `low`).

Finally, in Example 3 (Iterative declassification) notice that the static analyzer generates an iteration counting constraint for the declassification matching (Table IVa). The second constraint denotes that the statement at program point 4 must iterate at least 25 times. Also, notice that this constraint cannot be verified at preload time, so the checker marks this as `RT`, meaning it needs to be checked at runtime. As for the checklist to be passed to the runtime enforcer, for the only flow to be safe, the declassification constraints must be all satisfied. Based on that, the checklist for the runtime enforcer has two items (Table IVc): a request for counting the number of times a loop will run, and then using that number to validate the output operation.

On Algorithm 4, consider that $fr(C)$ is the flow report for program C, with each $f \in fr(C)$ being a single flow. For a flow f, $from(f)$ and $to(f)$ return the left-hand side (set of inputs and declassifications) and right-hand side (output), respectively. The same applies for a declassification d, with also $constr(d)$ denoting its constraint set. Also, when a label l is obtained from $getChannelLabel(cmd)$ (a system call that returns the label to the channel associated with cmd), an entry is made on $id(l)$ representing the original program point that generated the label. Command $compareChannel(cmd_1, cmd_2)$ is another system call that determines whether two commands represent access to a same I/O channel. This call may return `RT` if the check can only be made during runtime. For the entries of the runtime checklist ($chklst$), text in `typewriter` font represents the runtime enforcer directives, here treated as constant strings, whereas text in standard $math$ notation represents statements that are actually evaluated by the preloader algorithm. Function max appears with two different uses: on lines 13 and 18 it is used to calculate a maximum result for constraint checking, using the ordering `NO` $>$ `RT` $>$ `OK`; on lines 31 and 33 it is used over labels by the following: if the set of input labels only contain static labels (e.g., `low`, `high`), it evaluates to the most strict label (`high`); if, however, the set includes a dynamic label (`data` or `runtime`), it evaluates to `max(in_label(n),m)`, where `n` is the program point of the dynamic label and `m` the max of the static labels. In the latter case, `max` will be evaluated at runtime.

TABLE IV
PRELOAD CHECKER OUTPUT FOR EXAMPLE 3. (A) DECLASSIFICATION CONSTRAINTS; (B) LABEL TRANSLATIONS; (C) RUNTIME CHECKLIST

Constraint (X_1)	Status
(α_5, d)	OK
$(4, it \geq 25)$	RT
where $d = fetch(openDBConnection())$	

(a)

Flow labels
$\{\{\texttt{high}\} \mapsto^{\texttt{RT}} \texttt{low}\} \rightsquigarrow \texttt{high}$

(b)

Program point	Statement	Check condition
4	`while ... do`	`count_iter`
10	$output(\ldots)$	`eval(iter_count(4)` $\geq$ `25)`

(c)

About `data` *input channels inside loops*. Since our enforcement is not permissive (i.e., does not accept safe executions of possible unsafe programs), storing labels of `data` input channels inside loops can cause problems, as the label would be overwritten at every iteration of the loop. To solve this problem without permissiveness, our approach, when trying to store a preexisting `data` label, replaces the stored one with a "most strict join" of both. This can lead to imprecise analysis (i.e., over strict), but only in some rare cases: when a `data` input channel is read within a loop, with only some of its values (the lower labeled) being aggregated together (higher labeled ones being discarded), and then sent to an output. However, as explained in Section IX, our approach can be extended to be permissive, ruling out this imprecision.

VIII. THE RUNTIME ENFORCER

The runtime enforcer has a very simple behavior, similar to the one described in [21]. As the program is executed, each check on the checklist is performed as its corresponding program point is achieved. The runtime enforcer itself is a simple program, containing different functions for each type of check, and its own state-tracking variables. In this section we consider a Java-based runtime environment. Thus, our enforcer is a Java class with only *static* methods and parameters. Consequently, only a single instance of the enforcer is instantiated for a monitored program. Calls to the enforcer class are injected in the target application's bytecode, after the preload check, just before execution. Here we treat this code injection as a preliminary step to the runtime enforcement, although it can also be considered a final stage of the preload checker.

Code injection. The approach of injecting calls to the runtime enforcer in the application bytecode, just before execution, brings advantages for two reasons. First, it keeps the runtime enforcement stage with minimal overhead, as the injected code is a simple method call containing all information needed for that check. This precludes the need for the enforcer to monitor every single instruction, and to iterate over the different types of check. Second, it connects the program points calculated by the static analyzer (over the source code) with the information available to the runtime enforcer (which works on the bytecode).

Algorithm 4: Pre-load checker

```
1  chklst := ∅;
2  foreach f ∈ fr(C) do
3      fromLbl := ∅;
4      toLbl := getChannelLabel(cmd(to(f)));
5      foreach e ∈ from(f) do
6          if e ∈ (In × ℕ) then
           fromLbl ∪= getChannelLabel(cmd(e));
7          else if e ∈ Declass then
8              cmax := 0;
9              foreach x ∈ constr(e) do
10                 if x ≡ (i, exp) then
11                     chklst ∪= (i, count_iter);
12                     chklst ∪= (id(to(f)),
13                     eval(replace(exp, it, iter_count(i))));
14                     cmax := max(cmax, RT);
15                 else if x ≡ (α_i, n_d) then
16                     c := compareChannel(cmd(α_i), cmd(n_d));
17                     if c = RT then
18                         chklst ∪=
                           (i, compare_ch(cmd(n_d)));
19                     cmax := max(cmax, c);
20             if cmax ∈ {OK, RT} then fromLbl ∪= to(e);
21             else foreach α_i ∈ from(e) do
22                 fromLbl ∪= getChannelLabel(cmd(α_i));
23         foreach l ∈ fromLbl do
24             if l = data then
25                 chklst ∪= (id(l), store_data_label);
26                 chklst ∪= (id(to(f)), check_input(id(l)));
27             else if l = runtime then
28                 chklst ∪= (id(l), store_ch_label);
29                 chklst ∪= (id(to(f)), check_input(id(l)));
30             else if l ⊐ toLbl then return false;
31         if toLbl = data then
32             chklst ∪=
               (id(to(f)), set_data_label(max(fromLbl)));
33         else if toLbl = runtime then
34             chklst ∪=
               (id(to(f)), check_output(max(fromLbl)));
```

The injection is simple: for each check at the runtime checklist, a call for the enforcer to perform such a check is added just before the corresponding program point. We demonstrate the process via an example: consider a Java implementation of Example 1, shown in Fig. 2. Line 29, in bold font, represents the output command that needs to be checked. Fig. 3 shows a snippet of the corresponding `.dex` bytecode, compiled for Android's Dalvik virtual machine, already with the injected code, identified by the comment lines. Bold font is used to point the output instruction for which the check is needed. Note that in the "debug info" of the bytecode, it can be seen that bytecode address 0049 (recalculated from its original value, after the code injection) corresponds to program point 29, the program point where the output happens in the Java source code. Here, the check `check_output` of the enforcer is injected right before the output command. Since the code injection is a simple (and technology dependent) process, we omit a detailed specification of it.

The enforcer program. The enforcer provides a method for each check type. For each case, a statement is executed and its result validated. If the statement is not satisfied (i.e., expression does not hold, or command cannot be executed) then the enforcer halts the calling thread, and reports the violation.

```
...     ...
15:     static void processContactList() {
16:         String [] clist;
17:         String contact, text, addr;
18:         int counter, age;
19:         clist = getContactList();
20:         counter = 0;
21:         while(hasNext(clist)) {
22:             contact = getContact(clist);
23:             age = getAge(contact);
24:             if(age > 45)
25:                 counter = counter + 1;
26:         }
27:         text = "I have " + counter +
                    " contacts over age 45";
28:         addr = readFromInput();
29:     sendSMS(addr, text);
30:         System.out.println(text);
31:     }
```

Fig. 2. Java implementation for Example 1.

For the considered Java enforcer, each check is implemented by a method, e.g., `check_output(label)` is implemented by method $Enforcer.checkOutput(i, c, label)$, where i and c are arguments representing the current program point and command, respectively.

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 memory overhead, the enforcer keeps two buffers, $iter\_count$ and in_label, 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 $iter\_count$ point to looping and entries in in_label 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 n commands, exact $n-1$ 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 memory, 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 n 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 n of commands. But most practical cases do not reach the bound n, 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 [53], the runtime enforcer monitors *every* command of the program. Our enforcer, though, does not necessarily need to monitor every instruction, since the task

```
                        |[36c] Example1.processContactList:()V
                        ...
0003e4: 7100 0900 0000| 0034: invoke-static {}, Example1.readFromInput:()Ljava
                        |       /lang/String; // method@0009
0003ea: 0c01          | 0037: move-result-object v1
\\ Begin injected code
0003ec: 1302 1500     | 0038: const/16 v2, #int 21 // #0015
                        ...
000408: 7140 0100 3254| 0046: invoke-static {v2, v3, v4, v5}, Enforcer.checkOu
                        |       tput:(ILjava/lang/String;[Ljava/lang/Object;Ljav
                        |       a/lang/String;)V // method@0001
\\ End injected code
00040e: 7120 0a00 0100| 0049: invoke-static {v1, v0}, Example1.sendSMS:(Ljava/
                        |       lang/String;Ljava/lang/String;)V // method@000a
000414: 6201 0000     | 004c: sget-object v1, java.lang.System.out:Ljava/io/Pr
                        |       intStream; // field@0000
000418: 6e20 0b00 0100| 004e: invoke-virtual {v1, v0}, java.io.PrintStream.pri
                        |       ntln:(Ljava/lang/String;)V // method@000b
00041e: 0e00          | 0051: return-void
                        | debug info
                        |   line_start: 14
                        |   parameters_size: 0000
                        |   0000: prologue end
                        ...
                        |   0049: advance pc
                        |   0049: line 29
                        ...
```

Fig. 3. Dalvik bytecode snippet for a Java implementation of Example 1.

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 benchmarks we 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.

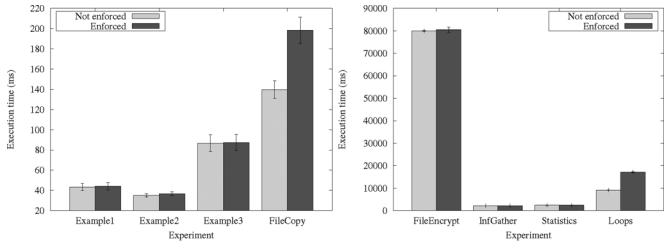
TABLE V
STATISTICS FOR BENCHMARK APPLICATIONS

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

For completeness, in Table V we report statistics about the size of applications used in our evaluation.

Each program was executed 50 times with and without the calls to the runtime enforcer, and their processing times and memory usage was observed. Fig. 4 presents the processing times of the programs. Error bars are for confidence intervals of 95%.

Note that only the "extreme" examples incurred a large processing overhead. In *FileCopy*, there is almost no processing between I/O accesses. The enforcer gets the data label from each input read, and applies it to each output write. Thus, the enforcer nearly does the same amount of computing as the original program itself. Notice how the enforcer overhead becomes minimal



(a) (b)

Fig. 4. Processing times of experiments.

TABLE VI
MEMORY USAGE RATIO OF EXPERIMENTS

Program	AllocCount	AllocSize
Example1	1,012	1,003
Example2	1,019	1,002
Example3	1,000	1,000
FileCopy	1,022	1,013
FileEncrypt	1,000	1,000
InformationGather	1,084	1,000
Statistics	1,253	1,001
Loops	3,413	1,001

when processing is added between the I/O accesses, in *FileEncrypt*. A similar thing happens in *Loops*, where the program is made entirely by loops, and the enforcer counts number of iterations on all of them. This way, the amount of injected code is large. In all other cases, overhead was almost imperceptible.

Table VI presents the results for memory usage. *AllocCount* and *AllocSize* represent number of memory allocations and used memory size, respectively. Each cell represents the ratio between the value for running that program with and without the enforcer. As expected, the overhead on used memory is minimal, being at most 1.3%, for the *FileCopy* program, in which labels are stored in every I/O access. For programs in which loop counting is done, the number of memory allocations can increase noticeably with the enforcer, as seen in *Statistics* and *Loops*. However, since for each loop only an integer is used to count, the overhead on used memory size is still minimal.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a hybrid static-runtime approach for enforcement of information flow and declassification policies. We presented examples, all within the context of mobile devices, that demonstrated the expressiveness of the policies our system can treat. By adding an intermediate step between static analysis and runtime enforcement, we managed to make both the static analysis independent of a particular system's security labels, and also the runtime enforcement be lightweight, consisting of only constant computation time checks. Our system supports unannotated code, due to the extension of the PCR analyzer, as well as declassification policies. Both the policies and the labeling system might include runtime constraints which are verified by the runtime enforcer. We have implemented the runtime enforcer, and shown via experiments that its overhead is negligible for most practical scenarios.

In order to simplify definitions, two assumptions were made in our mechanism. First, we consider that every declassification that needs to be checked during runtime is *necessary*. That is, if the declassification constraints are not satisfied at runtime, then the program is marked as unsafe without further analysis. This also means that nested declassifications do not need to be checked, as the failure of the outermost one will result in stopping the execution. This assumption can be relaxed by extending our mechanism so that the runtime checklist contains information of what to do when a declassification fails: stop the program, still allow it or perform further checks, depending on the labels. In favor of clarity, we leave such extension for future work. Second, the flow report contains all inputs that an output can *possibly* depend on. In other words, our runtime enforcement is not permissive to the point of accepting safe executions of potentially unsafe programs. Again, the mechanism can be extended so that runtime checks verify if unsafe branches are taken or not, but we leave this as future work.

Another important issue to be addressed in future work is a formal soundness proof for our solution. For this, a possible approach could be to use a different proving technique for each single step (e.g., a proof for our static component is already given in [46], by means of Policy Controlled Release (PCR) mechanism), and then prove the soundness of the composition of the different steps. We expect this proof to be a significant additional contribution to the current work, and we identify the main challenges and possible pitfalls in: i) modelling the complex system we are considering, i.e., providing a sound formalization of the Android system; ii) proving the soundness of putting together components that have been proved with different techniques.

Finally, as future work we also aim at investigating security features in aspect oriented programming [22], and whether policy specific enforcing code can be stitched as aspect programming.

REFERENCES

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *Proc. POPL'06*, 2006, pp. 91–102.
[2] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *Proc. SAS'04*, 2004, pp. 100–115.
[3] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers, "Sharing mobile code securely with information flow control," in *Proc. SP'12*, 2012, pp. 191–205.
[4] A. Askarov and A. Myers, "A semantic framework for declassification and endorsement," in *Proc. ESOP'10*, 2010, pp. 64–84.
[5] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *Proc. SP'07*, 2007, pp. 207–221.
[6] A. Askarov and A. Sabelfeld, "Localized delimited release: Combining the what and where dimensions of information release," in *Proc. PLAS'07*, 2007, pp. 53–60.
[7] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *Proc. CSF'09*, 2009, pp. 43–59.
[8] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. SIGPLAN Notices*, Dec. 2009, vol. 44, no. 20–31.
[9] J.-P. Banâtre, C. Bryce, and D. L. Métayer, "Compile-time detection of information flow in sequential programs," in *Proc. ESORICS'94*, 1994, pp. 55–73.
[10] S. Bandhakavi, W. H. Winsborough, and M. Winslett, "A trust management approach for flexible policy management in security-typed languages," in *Proc. CSF'08*, 2008, pp. 33–47.
[11] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a Java-like language," in *Proc. CSFW'02*, 2002, p. 253.
[12] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *Proc. SP'08*, 2008, pp. 339–353.
[13] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," *ACM Trans. Software Eng. Methodology*, vol. 18, pp. 9:1–9:43, Jun. 2009.

[14] S. Chong and A. C. Myers, "End-to-end enforcement of erasure and declassification," in *Proc. CSF'08*, 2008, pp. 98–111.
[15] S. Chong, A. C. Myers, K. Vikram, and L. Zheng, Jif Reference Manual, Jun. 2006.
[16] M. Conti, E. Fernandes, B. Crispo, and Y. Zhauniarovich, "CRePE: A system for enforcing fine-grained context-related policies on Android," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 5, pp. 1426–1438, Oct. 2012.
[17] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens, "Security monitor inlining for multithreaded Java," in *Proc. ECOOP 2009*, 2009, pp. 546–569.
[18] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Poster: Control-flow integrity for smartphones," in *Proc. CCS'11*, 2011, pp. 749–752.
[19] D. E. Denning, "A lattice model of secure information flow.," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976.
[20] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. OSDI'10*, 2010, pp. 1–6.
[21] U. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: A retrospective," in *Proc. NSPW'99*, 1999, pp. 87–95.
[22] G. Florez-Larrahondo and W. Haddock, "Aspect oriented programming with hidden markov models to verify design use cases," in *Proc. AOSD'09*, 2009, pp. 223–228, ACM.
[23] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. SP'82*, 1982, pp. 11–20.
[24] K. Hamlen, "Security Policy Enforcement by Automated Program-Rewriting," Ph.D. Thesis, Ithaca, NY, USA, 2006, AAI3227141.
[25] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Certified in-lined reference monitoring on .net," in *Proc. PLAS'06*, 2006, pp. 7–16.
[26] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 1, pp. 175–205, Jan. 2006.
[27] D. Hedin and A. Sabelfeld, "Information-flow security for a core of javascript," in *Proc. CSFW'12*, 2012, pp. 3–18.
[28] B. Hicks, D. King, P. McDaniel, and M. Hicks, "Trusted declassification: High-level policy for a security-typed language," in *Proc. PLAS'06*, 2006, pp. 65–74.
[29] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-mac: A run-time assurance tool for java programs," *Electron. Notes Theo. Computer Sci.*, vol. 55, no. 2, pp. 218–235, 2001.
[30] B. W. Lampson, "Protection," *SIGOPS Operat. Syst. Rev.*, vol. 8, no. 18-24, Jan. 1974.
[31] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, Dec. 1992.
[32] G. Le Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in *Proc. CSF'07*, 2007, pp. 218–232.
[33] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," in *Proc. ASIAN'06*, 2007, pp. 75–89.
[34] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theor. Comput. Sci.*, vol. 411, no. 19, pp. 1974–1994, Apr. 2010.
[35] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: Enforcement mechanisms for run-time security policies," *Int. J. Inf. Security*, vol. 4, no. 1-2, pp. 2–16, Feb. 2005.
[36] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of non-safety policies.," *ACM Trans. Inf. Syst. Security*, vol. 12, no. 19, pp. 1-19–41, Jan. 2009.
[37] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of non-safety policies," *ACM TISSEC*, vol. 12, no. 19, pp. 1-19–41, Jan. 2009.
[38] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results," in *Proc. ESORICS'10*, 2010, pp. 87–100.
[39] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proc. SSYM'05*, 2005, pp. 18–18.
[40] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. POPL'99*, 1999, pp. 228–241.
[41] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum, "A virtual machine based information flow control system for policy enforcement," *Electr. Notes Theor. Comput. Sci.*, vol. 197, no. 1, pp. 3–16, 2008.
[42] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *Proc. NDSS'07*, San Diego, CA, USA, 2007.
[43] F. Pottier and V. Simonet, "Information flow inference for ML.," *ACM Trans. Progr. Lang. Syst.*, vol. 25, no. 1, pp. 117–158, 2003.
[44] L. Qin, S. Duanfeng, H. Xinhui, and Z. Wei, "A hybrid security framework of mobile code," in *Proc. COMPSAC'04*, 2004, pp. 390–395.
[45] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, Sep. 1994.
[46] B. P. S. Rocha, S. Bandhakavi, J. den Hartog, W. H. Winsborough, and S. Etalle, "Towards static flow-based declassification for legacy and untrusted programs," in *Proc. SP'10*, 2010, pp. 93–108.
[47] G. Russello, M. Conti, B. Crispo, and E. Fernandes, "Moses: Supporting operation modes on smartphones," in *Proc. SACMAT'12*, 2012, pp. 3–12.
[48] G. Russello, M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Demonstrating the effectiveness of moses for separation of execution modes," in *Proc. CCS 2012*, 2012, pp. 998–1000.
[49] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proc. CSF'10*, 2010, pp. 186–199.
[50] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *Proc. ISSS'03*, 2003, pp. 174–191.
[51] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *Proc. Perspectives of Systems Informatics*, 2009, pp. 352–365.
[52] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Proc. CSFW'05*, 2005, pp. 255–269.
[53] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Security*, vol. 3, pp. 30–50, Feb. 2000.
[54] F. B. Schneider, J. G. Morrisett, and R. Harper, "A language-based approach to security," in *Proc. Informatics—10 Years Back. 10 Years Ahead*, 2001, pp. 86–101.
[55] A. Shinnar, M. Pistoia, and A. Banerjee, "A language for information flow: Dynamic tracking in multiple interdependent dimensions," in *Proc. PLAS'09*, 2009, pp. 125–131.
[56] P. Shroff, S. Smith, and M. Thober, "Dynamic dependency monitoring to secure information flow," in *Proc. CSF'07*, 2007, pp. 203–217.
[57] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *Proc. SP'08*, 2008, pp. 369–383.
[58] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proc. PLDI'09*, 2009, pp. 87–97.
[59] S. Tse and S. Zdancewic, "Run-time principals in information-flow type systems," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, Nov. 2007.
[60] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Computer Security*, vol. 4, pp. 167–188, 1996.
[61] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proc. ACSAC'12*, pp. 299–308.
[62] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proc. SOSP'09*, 2009, pp. 291–304.
[63] J. Yu, S. Zhang, P. Liu, and Z. Li, "Leakprober: A framework for profiling sensitive data leakage paths," in *Proc. CODASPY'11*, 2011, pp. 75–84.

Bruno P. S. Rocha photograph and biography not available at the time of publication.

Mauro Conti photograph and biography not available at the time of publication.

Sandro Etalle photograph and biography not available at the time of publication.

Bruno Crispo photograph and biography not available at the time of publication.