FASE: Functionality-Aware Security Enforcement

Petar Tsankov*
ETH Zurich
ptsankov@inf.ethz.ch

Marco Pistoia
IBM T. J. Watson Research Center
pistoia@us.ibm.com

Omer Tripp†
Google Inc.
trippo@google.com

Martin Vechev
ETH Zurich
martin.vechev@inf.ethz.ch

Pietro Ferrara†
Julia
pietro.ferrara@juliasoft.com

Dynamic information-flow enforcement systems automatically protect applications against confidentiality and integrity threats. Unfortunately, existing solutions cause undesirable side effects, if not crashes, due to unconstrained modification of run-time values (e.g. anonymizing sensitive identifiers even when these are used for authentication).

To address this problem, we present Functionality-Aware Security Enforcement (FASE), a lightweight approach for efficiently securing applications without breaking their functionality. The key idea is to let developers specify functionality constraints and then use a run-time synthesizer to replace sensitive values with constraint-compliant ones. Concretely, FASE consists of: (i) an efficient fine-grained data-flow-tracking engine, (ii) a domain-specific language (DSL) for expressing functionality constraints, (iii) a synthesizer that derives constraint-compliant values at security-sensitive operations, and (iv) an enforcement mechanism that automatically repairs illicit flows at run time.

We instantiated FASE to the problem of securing Android applications. Our experiments show that the FASE system is useful in practice: Its average run-time overhead is <12%; it avoids the crashes, side effects, and run-time errors exhibited by existing solutions; and the constraints in the FASE DSL are readable and concise.

1. INTRODUCTION

Improper enforcement of information-flow security remains the main cause of software vulnerabilities [22, 24]. This comes as no surprise—enforcing information-flow security is hard as it requires *global* reasoning about *transitive* information flows throughout the application as well as *subtle* checks and mutations of sensitive data.

*Parts of this author's contribution to this work took place while the author was a Research Intern at IBM's T. J. Watson Research Center.

†All of this author's contribution to this work took place while the author was employed by IBM Corporation at IBM's T. J. Watson Research Center.

ACSAC '16, December 05 - 09, 2016, Los Angeles, CA, USA

ISBN 978-1-4503-4771-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2991079.2991116

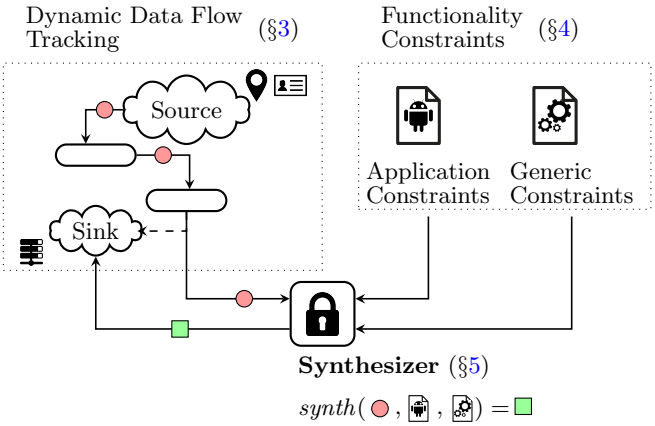


Figure 1: Functionality-Aware Security Enforcement

Beyond these general challenges, there are also factors that are specific to the mobile setting. Mobile developers have to cope with pressing demand for software releases and updates, leaving little room for security education and practice. At the same time, mobile platforms are rich in security-sensitive APIs—for example, for accessing device sensors and performing inter-application communication (IAC). Using these APIs in a secure manner requires special knowledge that developers often do not have.

Existing Work. The tension between the security needs of mobile software and the general lack of security savvy among developers motivates solutions for dynamic security enforcement. Recently, several promising approaches featuring low run-time overheads have been proposed; see e.g. [1, 18, 31]. However, existing solutions ignore constraints due to the application's functionality, and thus cause undesirable side effects and crashes [18]. For example, mocking an identifier (e.g., the device ID) may be acceptable if the mobile application sends it to a remote server for advertising or analytics purposes, but not if it is used for identifying the user. Fundamentally, securing an application dynamically without breaking it hinges on precisely understanding its functionality, which is difficult to uncover in a purely automated manner. Nonetheless, this is an essential requirement for the wide adoption of automated protection solutions.

Our Approach. We present Functionality-Aware Security Enforcement (FASE), a dynamic security enforcement approach designed for developers without security background. Our key insight is that combining fine-grained information-

flow tracking with a concise description of the application's functionality is sufficient to precisely detect, and correctly repair, information-flow vulnerabilities at run time. Developers declaratively specify functionality constraints in a designated DSL. The security enforcement system, in turn, automatically repairs information-flow vulnerabilities while satisfying all specified constraints.

We illustrate the overall flow and key components of FASE in Figure 1. The security goal is to protect applications against: (i) *confidentiality violations* (leaking private data to unauthorized parties, such as remote servers and other apps running on the device), and (ii) *integrity violations* (using unsanitized data in security-sensitive computations). Below, we briefly describe FASE's key components.

Fine-grained Information-flow Tracking. FASE features a dynamic fine-grained tracking engine that traces sensitive data *at the byte (rather than the object) level* as it flows from *sources* (i.e., methods that output private or untrusted data) to *sinks* (i.e. private data releasing points or security operations using untrusted input). Byte-level tracking is an essential prerequisite for precisely repairing illicit flows at run time for two reasons. First, dynamic data correction requires contextual information that is *only* available at the sink. Therefore, sensitive data cannot be anonymized/sanitized at the source. Second, data typically undergoes multiple transformations before reaching a sink. Therefore, data-flow tracking must be fine-grained to prevent benign data from being unnecessarily modified as this will likely cause undesirable side effects and crashes. For example, confidential data is often concatenated to URL strings. The system must modify only the confidential parts of the URL string because changes to the other URL parts (e.g. the host path) may result in unwanted side effects or even cause application crashes.

Two Kinds of Functionality Awareness. There are two kinds of functionality awareness. The first kind includes *generic* constraints imposed by the API, such as the preconditions of network and file-system APIs. For example, a URL string flowing into a network API must remain a well-formed URL string after the protection system modifies it. These constraints are general, and hence they are preconfigured in the security enforcement system. The second kind includes *application* constraints, such as the distinction between essential and extraneous servers to which identifiers are sent, which impose further constraints on the URL. As these constraints are specific to the application at hand, they are defined by the application's developers.

Domain-specific Language. FASE features a DSL that developers use to *declaratively* express application-specific functionality constraints. The constraints are purely functional—free of any security considerations—which creates a clean separation of concerns: the developers focus on the application's functionality, while the enforcement system augments the application's behavior with security aspects. We empirically show that our DSL is sufficiently expressive for capturing constraints arising in real-world apps while restricting the run-time overhead of constraint solving to a tolerable level.

Run-time Synthesis. Enforcing information-flow security amounts to replacing all sensitive data passed to sinks with constraint-compliant values; see Figure 1. Generating values that satisfy *all* functionality constraints—both generic and application-specific—is nontrivial as the constraints are often interdependent. Towards this, FASE features a specialized and highly-efficient synthesizer for string constraints. The FASE system utilizes the synthesizer to automatically generate values that satisfy all their associated constraints.

Contributions. Our main contributions are:

- A new approach, called FASE, for enforcing information-flow security at run time while preserving functionality.
- An efficient fine-grained information-flow-tracking engine for strings and primitive values, ensuring that the enforcement system modifies only sensitive data (§3).
- A domain specific language expressive enough to capture functionality constraints of real-world apps (§4).
- A synthesizer consisting of a specialized solver that is fast and scalable, and can precisely enforce the desired functionality constraints (§5).
- An implementation of FASE for Android featuring an efficient fine-grained information-flow-tracking engine. Our experiments with real-world applications indicate that (i) our DSL can capture interesting constraints, (ii) the FASE system is robust and does not cause functional side effects or application crashes, and (iii) the run-time overhead is less than 12% on average (§6).

2. OVERVIEW

In this section, we outline the FASE system. We do so with reference to an Android application that releases confidential data to the network. Afterwards, we discuss the challenges that must be accounted for when securing this application and we illustrate how FASE addresses them. Finally, we present our attacker model and state the security guarantees provided by the FASE system.

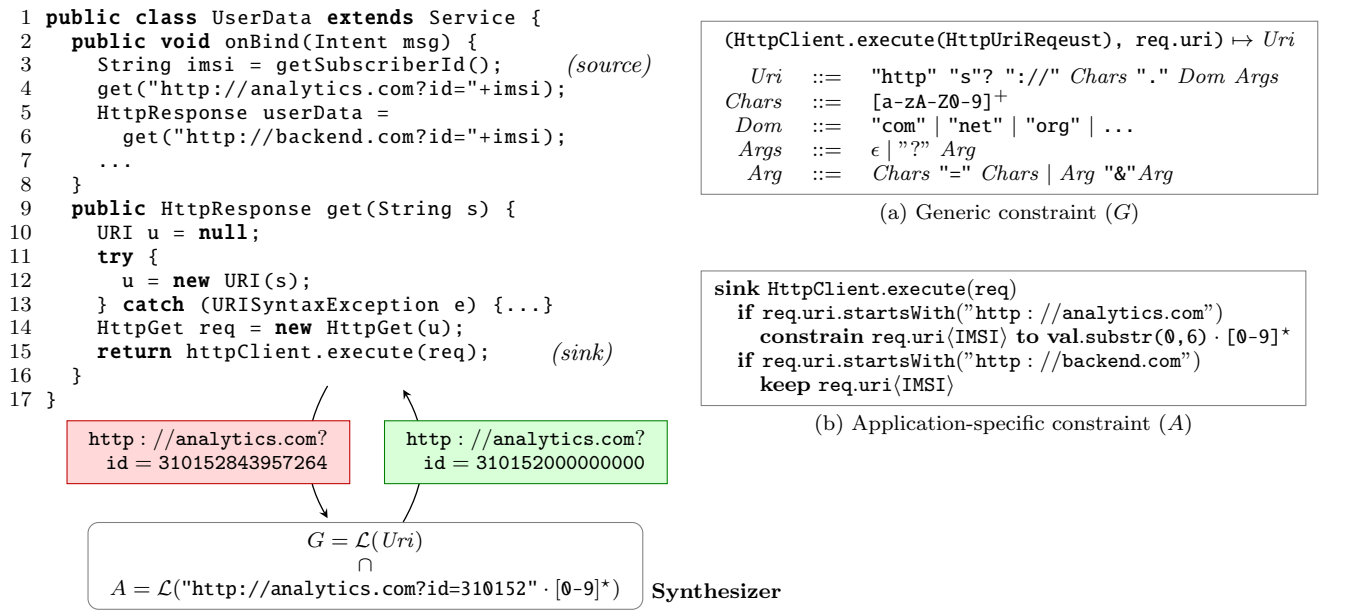
2.1 Motivating Example

The program shown in Figure 2(left) implements an Android service component for fetching user data from the application's backend server and for collecting user data for analytics. The service sends an HTTP request to the application's analytics server. This request contains the International Mobile Subscriber Identity (IMSI), obtained via a call to the `getSubscriberId` API. The analytics server uses only the first 6 digits of the IMSI, which are sufficient to extract the country code and the network code of the device. The service then creates a second request, this time passing the IMSI to the backend server, which uses the IMSI to identify the user's device. The Android service executes two security-sensitive operations: (i) sending the IMSI to the analytics server, and (ii) sending it to the backend server. As we illustrate next, in both cases, the enforcement system cannot naively anonymize the URI strings that contain the IMSI.

2.2 Security Enforcement Challenges

We next outline some of the key challenges that must be accounted for when securing the presented Android application.

Generic Constraints. Naively masking *all* confidential data to protect the user's privacy fails in practice because it may end up violating the application's overall functionality. In our example, one cannot replace the URI strings passed as parameters to `get` at lines 4 and 6 with arbitrary strings,



(a) Generic constraint (G)

(b) Application-specific constraint (A)

Figure 2: Securing an Android application using the FASE approach

because the `execute` API invoked at line 15, accepts only HTTP request objects with well-formed URI strings [27]. The security-enforcement system must ensure that such preconditions are satisfied to prevent run-time errors and application crashes. We refer to such preconditions as *generic constraints*.

We remark that such generic constraints are common for integrity sinks as well. For example, any modifications to an untrusted user input appended to an SQL query must preserve the well-formdness of the SQL query before it is passed to the `SQLiteDatabase.execQuery` integrity sink.

Application-specific Constraints. Satisfying generic constraints alone is, however, often insufficient for securing an application without causing side effects. For example, the application's functionality is disrupted if the security system completely anonymizes the IMSI. The security system must keep the IMSI intact when it is sent to the backend server, because this server uses the IMSI to identify the user's device. However, the security system can anonymize the last 9 digits of the IMSI when it is sent to the analytics server, because this server, for its purposes, should only use the first 6 digits and ignore the remaining 9 digits. We refer to such application-specific functional requirements as *application constraints*.

Sink Sensitivity. Modifying sensitive data correctly depends on the sink and the run-time values flowing into it. For example, anonymizing the IMSI in our example depends on the URL object passed to the sink; concretely, it depends on the host name, which identifies whether the IMSI is being sent to the backend or to the analytics server. It is therefore impossible to anonymize the IMSI directly at the source (i.e., at line 3).

2.3 The FASE System

FASE automatically secures the application by guaranteeing that any confidential/untrusted data is anonymized/sanitized before being used in a sink, while enforcing the functional constraints imposed by the application and its library dependencies. We now illustrate how the motivating example is secured using the FASE system without breaking its functionality.

Fine-grained Information-Flow Tracking. The FASE tracking engine assigns the label `IMSI` to each byte (or, character) of the string returned by the source API call at line 3. Byte-level data flow tracking is key to precisely anonymize sensitive data because the data typically undergoes multiple transformations before reaching the sink. FASE propagates the label `IMSI` as the application appends the `imsi` string to construct the two URI strings at lines 4 and 6, and later when it constructs `URI` objects at line 12 and HTTP requests at line 14. FASE inspects the HTTP requests passed to the sink API at line 15. The application invokes the sink twice, and in both cases the passed HTTP request objects point to the labeled `imsi` string. The data flow tracker reports precisely which bytes of the URI strings represent the IMSI.

Functionality Constraints. To avoid side effects, the protection system is configured with generic constraints and application-specific constraints. The relevant constraints for our motivating example are shown in Figure 2(a-b).

The generic constraint in Figure 2(a) defines the precondition of the `execute` API method. This constraint formalizes, using the context-free grammar with start symbol *Uri*, that the URI strings in HTTP requests must indeed be well-formed URI strings. Such generic constraints are preconfigured in our FASE system, as they are not application-specific.

Developers capture application-specific functionality constraints using the FASE DSL. These constraints enable developers to designate the sensitive values that are important for the application's functionality. Intuitively, these constraints restrict how the FASE system modifies strings that originate from a source and flow to a sink. For example,

the first rule Figure 2(b) imposes that whenever the application sends the IMSI to the analytics server, FASE must not modify its first 6 characters, while it can arbitrarily replace the remaining 9. This is specified via regular expression **val**.`substr(0,6)` · `[0-9]`*, where **val** is bound to the `imsi` string at run time. The second constraint imposes that the IMSI must not be modified when it is sent to the backend server. This constraint is needed because the backend server uses the IMSI to identify the user's device. Both constraints restrict changes to the URI strings of HTTP requests. The FASE DSL also supports constraints that restrict modifications to sensitive values passed as parameters of HTTP POST requests. We give an example of such an application-specific constraint in Section 4.2.

Synthesizer. Any sensitive data passed to a sink is automatically repaired by calling a synthesizer at run time. The synthesizer guarantees that the anonymized/sanitized values satisfy all functionality constraints. By default, the synthesizer forbids all explicit flows.

Figure 2 shows how the URI string used to connect to the analytics server is anonymized using the synthesizer. For illustration, we fix the `imsi` string to `"310152284395726 4"`. Based on the concrete URI string at run time, the synthesizer derives a regular expression r from the application constraints A. The URI strings contained in this regular expression are indeed precisely those that keep the host name as well as the first 6 digits of the IMSI intact. Finally, the synthesizer intersects the language (denoted by $\mathcal{L}(\cdot)$) of the derived regular expression r with the language of the sink's context-free grammar *Uri*, a task known to be decidable, and returns a value that is contained in both languages.

2.4 Assumptions and Security Guarantees

We now describe our system and threat models, and then we state the security guarantees provided by FASE.

System Assumptions. We consider benign (as opposed to malicious) applications. These applications may have sensitive data flows, both legitimate (e.g. sending the IMSI to identify the user's device) and illegitimate (e.g. untrusted third-party applications sending values that are used to construct SQL queries and advertising libraries exfiltrating private data). To secure their applications, developers specify application-specific constraints to (i) identify the legitimate flows and to (ii) restrict changes to sensitive data over these flows. The latter is necessary because, as we have illustrated with our example, unconstrained modifications over legitimate flows may disrupt functionality. We remark that we intentionally ask developers to write only functionality constraints (as opposed to security rules), as they generally understand the functionality of their applications better than security.

Threat Model. We consider an adversary who can (i) observe sensitive values output to confidentiality sinks and (ii) can inject inputs at integrity sources. Our adversary can, for example, observe data that is sent as part of HTTP GET and POST requests over network APIs. Furthermore, our adversary can inject values through integrity sources such as inter-application communication APIs and network APIs. The latter may allow the attacker to crash the application by injecting malformed inputs, and to even compromise the integrity of sensitive data stored by the application. For example, the Google+ Android application was vulnerable to SQL injection attacks, allowing third-party applications to modify the application's database.

We remark that we focus on explicit data flows. Therefore, malicious applications that exfiltrate sensitive data over covert channels and implicit flows fall outside of our scope.

Security Guarantees. The FASE system, by default, masks all explicit flows unless the developer's constraints restrict such modifications. Therefore, the FASE system guarantees that any sensitive data passed over illegitimate explicit flows is masked with values that satisfy the functionality constraints.

Finally, we remark that our assumptions are consistent with similar state-of-the-art data flow security systems. Both static and dynamic data flow protection solutions, such as [2, 10], are easily bypassed by malicious applications.

3. FINE-GRAINED INFORMATION-FLOW TRACKING

In this section, we describe the FASE algorithm for fine-grained information-flow tracking.

3.1 Basic Notions and Notation

We begin by introducing several supporting notations and definitions. In our semantics, we partition the live values into disjoint sets *Objs* of *objects* and *Prims* of *primitive values*. Let $Strs \subseteq Objs$ be the set of (live) *string* objects. We denote by $\langle c_1 c_2 \cdots c_n \rangle$ the sequence of characters comprising a given string s, and by $len(s)$ the length of s. Given indexes i and j, such that $1 \leq i \leq j \leq len(s)$, we let $s[i]$ denote character c_i and $s[i, j]$ the substring $\langle c_i \cdots c_j \rangle$.

We also fix a set *Labels* of source *labels*. Intuitively, labels reflect two types of security-sensitive information: private values (such as the IMEI, IMSI and location, as in Figure 2) and untrusted inputs (such as those emanating from the Internet, IPC messages, etc.).

Finally, we fix a set $API = \{m_1, m_2, \ldots, m_h\}$ of method signatures. Methods that output security-relevant data (i.e., either private or untrusted data) are called *sources*, and are defined by $Sources \subseteq API$. A given source returns a particular label from *Labels*, as defined by the function $labelType\colon Sources \to Labels$. Methods that perform security-sensitive operations (i.e., either a data release or a sensitive computation) are referred to as *sinks*, and are defined by $Sinks \subseteq API$.

3.2 Instrumentation

FASE performs information-flow tracking over strings and primitive values. We do not explicitly track labels over objects. Instead, we track object labels indirectly by tracking strings and primitive values that are (transitively) reachable from the object's fields. The labels assigned to an object are then the union of the labels assigned to transitively reachable strings and primitives.

More precisely, FASE instruments the concrete state with function $\tau : Objs \cup Prims \to \mathcal{P}(Labels)$, which maps objects (including string objects) and primitive values to labels. For object o, $\tau(o)$ returns the set of labels assigned to the strings and primitive values pointed-to by the object o through a sequence of zero or more field dereferences.

FASE implements two different label tracking strategies: character-level tracking for strings and value-based tracking for primitive values. We explain these strategies below.

Character-level Tracking for Strings. Given a source method m, such that $labelType(m) = l$, let $s = \langle c_1 c_2 \cdots c_n \rangle \in Strs$ be a string object pointed-to by the object returned by m. (For sources that return a string, the returned object is s itself.) Via platform-level instrumentation, FASE maps each character $c_i \in \{c_1, \cdots, c_n\}$ to the label l, and therefore we have $\tau(s) = \{l\}$.

To precisely propagate the labels associated with individual characters, FASE instruments the Android implementation of all string operations. As an illustration, concatenation of strings $s = \langle c_1 c_2 \cdots c_n \rangle, s' = \langle c'_1 c'_2 \cdots c'_{n'} \rangle$, with $\tau(s) = \{l\}, \tau(s') = \{l'\}$, yields $s'' = \langle c_1 c_2 \cdots c_n c'_1 c'_2 \cdots c'_{n'} \rangle$, with $\tau(s'') = \{l, l'\}$. FASE maintains a label for each character, and so the first n characters of s'' have label l, while the remaining n' have label l' (i.e., $\tau(s''[1, n]) = \{l\}$ and $\tau(s''[n+1, n+n']) = \{l'\}$). Other string operations, such as `substring` and `replace`, are defined analogously.

Value-based Tracking for Primitives. Tracking primitives is needed for confidentiality, such as anonymizing the device's location, which is stored using doubles. The tracking engine implements a value-based strategy for primitive values, motivated by the insight that sensitive primitive values are mostly unique [31] (e.g., the latitude and longitude are 64-bit primitives.) Given a source $m \in Sources$, such that $labelType(m) = l$, let $v \in Prims$ be a primitive value reachable from the object returned by m. Using application instrumentation, FASE updates the map $\tau[v \mapsto \{l\}]$, which tracks the labels assigned to primitive values at run-time.

Our experience with real-world applications suggests that primitive values are often leaked as part of string values (e.g., as part of the URL query string). The engine instruments operations, e.g. `StringBuffer.append(double)`, that insert primitives into strings. That is, given primitive value v carrying label l, with string representation s_v, if s_v is appended to string s, then FASE assigns label l to each character of s_v in the resulting string $s \cdot s_v$.

Our empirical experience further indicates that operations on security-relevant primitives, such as addition and multiplication, are rare in practice. Hence, we intentionally avoid instrumenting them to retain low overhead.

Example. We conclude by illustrating the behavior of the FASE tracking engine. The example in Figure 3a motivates the handling of primitive values, the tracking of their conversion to strings, as well as the character-level precision for strings. In Figure 3b, we show how FASE assigns and propagates labels for this code.

At line 1, the returned `Location` object points to the primitive values that represent the location's latitude and longitude. Suppose these values are 37.3876 and 122.0575. To label these values, FASE updates the map τ as follows:

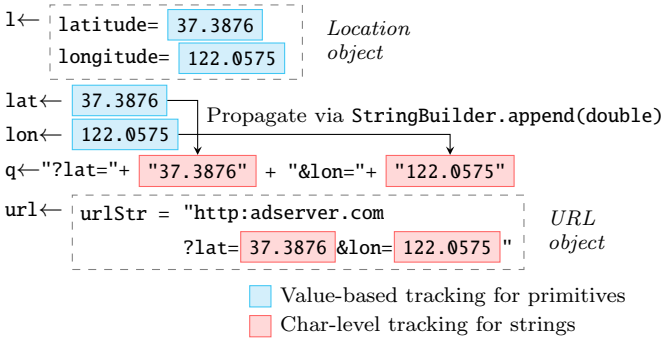
$$\tau \leftarrow \tau[\texttt{37.3876} \mapsto \{\texttt{Location}\}, \texttt{122.0575} \mapsto \{\texttt{Location}\}]$$

At lines 2 and 3, the labeled primitive values are stored into variables `lat` and `lon`. Note that FASE has already assigned the appropriate labels to these primitive values.

At line 4, the primitive values stored at `lat` and `lon` are appended to a string. To propagate their respective labels to the constructed string, FASE instruments the `append` method and propagates labels only to the characters corresponding to the appended primitive values. The constructed string is `"http://adserver.com?lat=37.3876&lon=122.0575"`. FASE labels the substrings `"37.3876"` and `"122.0575"` with `Location`.

```
1 Location l = getLastKnownLocation(GPS);
2 double lat = l.getLatitude();
3 double lon = l.getLongitude();
4 String q = "?lat=" + lat + "&lon=" + lon;
5 URL url = new URL("http://adserver.com"+q);
```

(a) Source code of the example



(b) Dynamic information-flow tracking

Figure 3: A common example of an information flow from a `Location` source to a network sink

At line 5, the `URL` constructor takes the constructed string `"http://adserver.com?lat=37.3876&lon=122.0575"`, where the characters of the substrings `"37.3876"` and `"122.0575"` are those marked with the label `Location`.

4. EXPRESSING FUNCTIONALITY CONSTRAINTS

We now describe the specification of generic constraints, and we present a DSL for application constraints. We consider string-related constraints as string is the predominant data type in the context of security and privacy. Examples includes strings received from untrusted users or third-party applications, which may compromise the integrity of the application's data, as well as strings that contain confidential information that the application exports.

4.1 Generic Constraints

We refer to the sinks' preconditions, which are defined by API designers, as *generic constraints*. These are defined by the function

$$G : Sinks \times Vars \rightarrow CFGs$$

where $Vars$ is a set of variables and $CFGs$ is the set of context-free grammars over a standard alphabet. A context-free grammar $g = G(snk, x)$ defines that the string stored at the variable x must be in $\mathcal{L}(g)$, where $\mathcal{L}(g)$ denotes the language of g. Unlike the application constraints, defined shortly, the generic constraints do not depend on the values passed to the sink and on the current application state. This is expected as they reflect constraints imposed by the API, not by the application.

As an example, consider the generic constraint:

$$(\texttt{SQLiteDatabase.execSQL}, \texttt{sql}) \mapsto \texttt{SQL} \ ,$$

where `SQL` is the CFG that defines the set of well-formed SQL queries; see Figure 2. This constraint specifies that the SQL query string, stored at the variable `sql`, must be in

$$
\begin{array}{lll}
\textit{Constrs} & ::= & \textit{Constr}, \ldots, \textit{Constr} \\
\textit{Constr} & ::= & \textbf{sink}\ \textit{name}\ \textbf{if}\ \textit{Cond}\ \textit{Expr} \\
\textit{Cond} & ::= & \textit{bool\_op}(\textit{arg}, \cdots, \textit{arg}) \\
 & | & (\textit{Cond}\ \textbf{and}\ \textit{Cond}) \mid (\textbf{not}\ \textit{Cond}) \\
\textit{Expr} & ::= & \textbf{constrain}\ \textit{var}\langle \textit{label} \rangle\ \textbf{to}\ \textit{Regex} \\
\textit{Regex} & ::= & \textit{SymbStr} \mid \textit{Regex}^* \mid \textit{Regex} \cdot \textit{Regex} \\
 & | & \textit{Regex} + \textit{Regex} \\
\textit{SymbStr} & ::= & \textit{str} \mid \textit{var} \mid \textbf{val} \mid \textit{str\_op}(\textit{arg}, \cdots, \textit{arg})
\end{array}
$$

Figure 4: BNF of the FASE DSL: $\textit{name} \in \textit{Sinks}$, $\textit{var} \in \textit{Vars}$, $\textit{label} \in \textit{Labels}$, $\textit{str} \in \textit{Strs}$, $\textit{arg} \in \textit{Vars} \cup \textit{Strs}$.

the language $\mathcal{L}(\texttt{SQL})$. Note that our motivating example is an object-oriented Android program, and `req.uri` represents the variable `req` followed by the field identifier `uri`. For simplicity, we do not refine variables into variables followed by (zero or more) field identifiers.

We note that the generic constraints are application independent. That is, they are defined once for each sink, regardless of how different applications use that sink.

4.2 Application-specific Constraints

We now define the FASE DSL for specifying application-specific constraints.

Requirements. We start with the abstract requirements that guided the design of the FASE DSL.

Sink Sensitivity: The constraints usually depend on *where* a sensitive value is used. For example, the constraints stipulating how sensitive values are handled when they are used in a database API may differ from the constraints for a network API.

Source Sensitivity: The constraints are *source sensitive*: the correct handling of sensitive values depends on where the values come from. For example, a device identifier is usually handled differently than the device's location.

State Sensitivity: The constraints often depend on the values passed to the sink. The application constraints of our motivating example, for instance, depend on HTTP request's host name and on the concrete IMSI value concatenated to the URL string.

Byte-level Granularity: The constraints often pertain to subsets of data. For example, the second application constraint of our motivating example states that the first 6 digits of the IMSI must not be modified while the last 9 digits can be anonymized.

Next, we define our FASE DSL, which is adequate with respect to these requirements.

Syntax and Semantics. The FASE language syntax is given in Figure 4. An application constraint has the form **sink** *name* **if** *Cond Expr*, where *name* is the signature of a sink method. A constraint conditionally restricts changes to strings passed to the sink.

A condition $\textit{bool\_op}(\textit{arg}, \cdots, \textit{arg})$ is evaluated with respect to the current state. Here *bool_op* is a pure method returning a boolean value. The arguments *arg* can be constant values, such as strings, or variables, which are resolved at run time. An application constraint is applicable to the current program state if its condition evaluates to true, otherwise it is ignored. For example, the constraint

sink `HttpClient.execute(req)`
if `req.uri.startsWith("http://analytics.com")` *Expr*

is applicable only to HTTP requests to the analytics server. Atomic conditions can be composed with the standard conjunction (**and**) and negation (**not**) connectives. Additional boolean connectives can be derived in the standard way.

The expression $x\langle l \rangle$, where x is a variable and l a label, returns a substring s of the string stored at x such that all characters in s are labeled with l and the characters wrapped around s are not labeled with l. That is, the substring s is a block of characters uniformly labeled with l. For example, if the variable `sql` points to the SQL string `"SELECT name, ph_number FROM contacts WHERE id=10 OR 1=1"`, where the substring `"10 or 1=1"` is marked with the label `UNTRUSTED`, then $\texttt{sql}\langle\texttt{UNTRUSTED}\rangle$ returns `"10 or 1=1"`. Note that a string may contains multiple such blocks.

The expression **constrain** $x\langle l \rangle$ **to** r introduces a constraint that restricts modifications to a substring labeled with l: All substrings $x\langle l \rangle$ must be replaced with strings from the language of the regular expression r. For example, consider the expression **constrain** $\texttt{req.uri}\langle\texttt{IMSI}\rangle$ **to** $\texttt{[0-9]}^\star$ and let `req.uri` point to the string `"imsi=0123"` where `"imsi="` is not labeled and `"0123"` is labeled with `IMSI`. This expression restricts modifications to `req.uri` to strings from the language $\mathcal{L}(\texttt{"imsi="} \cdot \texttt{[0-9]}^\star)$.

The FASE DSL features symbolic regular expressions which are resolved at run time. These are constructed out of variables, the keyword **val**, and pure methods *str_op* that return strings. The keyword **val** in the regular expression of a constraint **constrain** $x\langle l \rangle$ **to** r returns the substring $x\langle l \rangle$. Referring to the labeled substring is needed for regular expressions whose definition depends on the labeled substring, as illustrated in our example of Section 2. Using variables and the keyword **val**, developers can thus write regular expressions that are semantically resolved at run time. For example, the constraint

constrain $\texttt{req.uri}\langle\texttt{IMSI}\rangle$ **to val**$\texttt{.substr(0,6)} \cdot \texttt{[0-9]}^\star$

formalizes that the first 6 characters of the IMSI must be kept intact, while the remaining characters can replaced. We will illustrate the semantics of this constraint with an example shortly.

To simplify the writing of constraints, we introduce several simple syntactic shorthands: we write **sink** $mc_1e_1c_2e_2$ for the two constraints **sink** mc_1e_1 and **sink** mc_2e_2. We also write **keep** $x\langle l \rangle$ for **constrain** $x\langle l \rangle$ **to val**, which formalizes that substrings labeled with l must be kept intact. Note that we have used these shorthands to write the application-specific constraints for our motivating example in Section 2.

The semantics of an application-specific constraint A, denoted by $[\![A]\!]$, is defined as a function mapping an application state σ, a sink method *snk*, and a sink variable x to a regular expression $r = [\![A]\!](\sigma, \textit{snk}, x)$. Note that application-specific constraints depend on the current state σ because the constraint's condition, which defines whether the constraint is applicable or not, as well as all variables that appear in symbolic regular expressions are evaluated with respect to the current state. A string s *satisfies* the application-specific constraint A for the given state σ, sink *snk*, and variable x, if $s \in \mathcal{L}([\![A]\!](\sigma, \textit{snk}, x))$, where $\mathcal{L}(\cdot)$ denotes the language of the regular expression. Note that a set of application-

specific constraints is satisfied iff the string s is in the intersection of the languages defined by the constraints' regular expressions. Due to the technical nature of the FASE DSL's semantics, we relegate the formal treatment of the semantics to Appendix B. To describe the semantics of the language, we give a simple, but illustrative, example below.

Example. Consider the application-specific constraint:

```
sink HttpClient.execute(req)
  if req.uri.startsWith("http://analytics.com")
    constrain req.entity.content⟨IMSI⟩
      to val.substr(0,6) · [0-9]*
```

This constraint restricts modifications to the entity of certain HTTP POST requests. Similarly to the constraint from our motivating example given in Figure 2, this constraint is applicable to states where the string stored at `req.uri` starts with `"http://analytics.com"`. The above constraint formalizes that any substring stored at `req.entity.content` and labeled with `IMSI` must be replaced with a string from the regular language defined by **val**.`substr(0,6)` · `[0-9]`*. Note that while in our motivating example the application-specific constraint restricts modifications to URL strings, which in the case of HTTP GET requests often contain sensitive data, here the constraint restrict changes to sensitive values passed as parameters via HTTP POST requests. To illustrate, suppose `req.entity.content` points to `"id=310152843957264"`, where the substring `"310152843957264"` is labeled with `IMSI`. According to the semantics, the substring `"310152843957264"` is replaced by a string from the language of symbolic regular expression **val**.`substr(0,6)` · `[0-9]`*. which evaluates to the concrete regular expression `"310152"`·`[0-9]`*. The remaining characters are not labeled and thus remain unchanged. The derived regular expression is `"id=310152"`·`[0-9]`*. The string `"id=310152000000000"`, for example, satisfies this application constraint, while the string `"id=310152xxxxxxxxx"` does not.

4.3 Enforcement

Here we define the notion of correct security enforcement with respect to generic and application-specific constraints. Consider an app that calls a sink method *snk* in a given state σ. Imagine that the string s, pointed-to by the variable x, is labeled. The security enforcement system replaces s with s'. The enforcement is *correct* iff s' satisfies the generic constraint G and the application-specific constraints A, i.e. s' is in the language of the CFG $g = G(snk, x)$ as well as in the language of the regular expression $r = [\![A]\!](\sigma, snk, x)$. The reduces to checking their intersection: $s' \in \mathcal{L}(g) \cap \mathcal{L}(r)$. We detail the FASE synthesizer for correct enforcement in Section 5.

Discussion. We remark on several key points about the FASE DSL. We deliberately designed the language to support regular application constraints and context-free generic constraints. The intersection between a regular and a context-free language is context-free, which guarantees that membership is decidable. Note that using context-free languages for all constraints breaks decidability [17].

We opted for CFGs for sinks' preconditions as they are more subtle to encode. For example, the precondition of the sink `SQLiteDatabase.execSQL(String)`, which accepts only well-formed SQL queries, is not regular.

Algorithm 1: The main steps of synthesizer. All labeled values passed to a sink *snk* are replaced with constraint-compliant values.

Input: Generic constraints G,
Application constraints A,
Sink signature $snk^k(x_1, \ldots, x_k)$,
State σ
Output: Repaired state σ

```
1  begin
2    L ← []
3    for x ∈ {x_1, ..., x_k} do
4      if τ(x) ≠ ∅ then
5        r ← [[A]](σ, snk, x)
6        dfa ← convert r to DFA
7        g ← G(snk, x)
8        cfg ← intersect dfa with g
9        s ← generate a string from cfg
10       L ← L + [(x, s)]
11   for (x, s) ∈ L do
12     store s at x
13   return σ
```

We can approximate more complex context-free application constraints using regular approximations [20, 21]. For instance, one can under-approximate a context-free application constraint g with a regular expression r. This guarantees that any string that satisfies r also satisfies g, simply because $\mathcal{L}(r) \subseteq \mathcal{L}(g)$.

5. SYNTHESIZER

We describe our specialized synthesizer for strings, characterize its properties, and discuss several optimizations.

The synthesizer is invoked immediately before the application invokes a sink. For the example of Figure 2, it is invoked immediately before calling the sink API method `HTTPClient.execute()`. The synthesizer replaces each sensitive value passed to the sink with a value that satisfies all functionality constraints—both generic and application-specific.

We illustrate the main steps of the synthesizer in Algorithm 1. Our synthesizer is configured with the generic constraints G and a set A of application constraints expressed in the FASE DSL. The sink signature $snk(x_1, \ldots, x_k)$ identifies all variables that may point to labeled strings at run time. The synthesizer computes a new value for each variable $x \in \{x_1, \ldots, x_k\}$ that stores a sensitive value according to τ. Note that x stores a sensitive value if the labeling function τ maps x to a nonempty set of labels. To synthesize a new value, the synthesizer first derives the regular expression r from the application constraints A, as defined in the FASE DSL semantics (line 5). Then, it converts the derived regular expression to a DFA (line 6) and intersects this DFA with the CFG $G(snk, x)$ of the sink (line 8). Here, our synthesizer leverages that the intersection of a context-free and a regular language is a context-free language [17] to compute the conjunction of all constraints. If this intersection is empty, then the synthesizer aborts and reports about the inconsistency. Otherwise, the synthesizer deterministically derives a string s from the computed CFG, generating the

replacement string for the variable x. The synthesized string s for the variable x is added to the list L (line 10). Finally, the synthesizer updates the variables with the synthesized strings (lines 11–12).

Key Points. There are several points of interest. First, the synthesizer, by default, forbids all explicit flows from sources to sinks, unless the application constraints restrict this. In more detail, without any application constraints, the synthesizer generates a fixed value that satisfies the sink's precondition and that does not depend on the actual sensitive value. Second, although the worst complexity for translating a regular expression of size n to a DFA (line 6) is exponential, our experimental results show that the regular expressions used in practical constraints have small DFA representations. Third, the problem of computing the intersection between a DFA and a context-free grammar is tractable. Finally, generating a string from the computed grammar takes linear time.

Optimizations. We conclude with two optimizations that address expensive computation steps, thereby improving the synthesizer's performance.

The regular expressions r derived from application constraints often imply the generic constraint g, i.e. $\mathcal{L}(r) \subseteq \mathcal{L}(g)$. Common cases are sensitive strings in URL strings such as `"http://a.com?arg=<sensitive>"` where application-specific constraints restrict changes to `<sensitive>` to alphanumerical strings. Any string satisfying the application constraints is a well-formed URL string that satisfies the generic constraint.

Deciding whether $\mathcal{L}(r) \subseteq \mathcal{L}(g)$ is as expensive as computing $\mathcal{L}(r) \cap \mathcal{L}(g)$. We thus select a string $v \in \mathcal{L}(r)$ and then check $v \in \mathcal{L}(g)$. If $v \notin \mathcal{L}(g)$, the synthesizer falls back to the original approach and computes the intersection.

As a further optimization, we can cache all computed context-free grammars used for deriving constraint-compliant strings. This works well as many labeled strings differ only in their labeled parts.

6. EXPERIMENTAL EVALUATION

We now present the experiments conducted to evaluate the design choices governing FASE and its effectiveness.

6.1 Implementation

The FASE system implements our fine-grained information-flow tracking engine (Section 3) and our synthesizer for generating values satisfying generic and application-specific constraints (Section 5).

To implement the character-level tracking for strings, the FASE system instruments the libraries `String`, `StringBuffer`, `StringBuilder`, and `AbstractStringBuilder`. To keep the overhead incurred by tracking low, we use spatial locality and store labels in the same character array used to store string characters. Since the `String` class is declared as final, instrumentation at the application-level based on inheritance is not possible.

The value-based tracking for primitive values is implemented using application-level instrumentation. FASE injects code for maintaining the key/value table that maps primitive values to labels. The instrumentation augments calls to methods such as `StringBuffer.append(double)` with code for propagating labels, as explained in Section 3.2.

The application-level instrumentation also injects bytecode for intercepting the calls to all sources and sinks in the FASE configuration, and insert calls to the synthesizer (Section 5), which we implemented in Java.

Technically, our instrumentation scheme is defined atop the ASM 5.0 framework [3] and the Dex2Jar tool [9], which transforms Dalvik code into Java bytecode. We used the ACLA framework [6, 7] to implement the synthesizer.

Configuration	Byte-level Tracking	Generic Constraints	App-specific Constraints
Coarse Tracking	✗	✗	✗
No Constraints	✓	✗	✗
Generic constraints	✓	✓	✗
All functional constraints	✓	✓	✓

Table 1: FASE features used by each configuration.

6.2 Engineering Requirements

We evaluate the FASE system with respect to three engineering requirements:

(R1) Robustness. The FASE system must secure applications without causing crashes, run-time errors, or other visible side effects.

(R2) Overhead. The overall overhead caused by the FASE system must be low.

(R3) Constraint Conciseness. Application constraints for real-world applications can be expressed concisely in the FASE DSL.

6.3 Experimental Setup

There have been numerous studies on popular Android applications with sensitive information flows; see e.g. [10, 18, 31]. As we needed precisely such applications for our experiments, we randomly selected applications from the experimental suites of [18] and [31]. Statistics about the selected applications are given in Figure 8 of Appendix A. We remark that many applications have more than a thousands sinks and all applications have sensitive flows. Common sensitive flows include confidentiality-relevant flows such as sending private data sent over HTTP GET and POST requests as well as integrity-relevant flows such as opening files with unsanitized file paths and executing unsanitized SQL queries. To gather a more complete data on the behavior of each of the applications, we wrote scripts to simulate user interaction, which exercise the applications more thoroughly. To write the scripts we had a user manually click through the application and record all the user's actions in the script.

We configured FASE with the set of sensitive sources and sinks specified in [25]. We conducted all the experiments using a Nexus 7 device running a modified Android with support for the FASE information-flow tracking engine.

6.4 Results

In the following, we present several sets of results and discuss our engineering requirements.

R1: Robustness. To obtain a better understanding of which features are needed for repairing sensitive information flows without causing side effects, we ran the selected applications using four different configurations:

App Name	Coarse Tracking			No Constraints			Generic Constraints			All Constraints (**FASE**)		
	Crash	Side-effects	Errors	Crash	Side-effects	Errors	Crash	Side-effects	Errors	Crash	Side-effects	Errors
Candy Crush Saga	○	○	0	○	○	0	○	○	0	○	○	0
Yellow Pages	●	●	5	○	○	2	○	○	2	○	○	0
Paper Toss	●	●	1	●	●	1	●	●	1	○	○	0
Smiley Pops	○	○	1	○	○	1	○	○	1	○	○	0
Coffee Finder	○	◐	3	○	◐	3	○	◐	3	○	◐	1
Bump	○	○	0	○	○	0	○	○	0	○	○	0
iHeartRadio	●	●	1	●	●	1	●	●	1	○	○	0
SmartTacToe	○	○	0	○	○	0	○	○	0	○	○	0
AccuWeather	●	●	2	○	●	2	○	●	2	○	○	0
aiMinesweeper	○	○	0	○	○	0	○	○	0	○	○	0
Antsmasher	●	●	4	●	●	4	○	◐	1	○	○	0
Cat Hair Saloon	○	◐	0	○	◐	1	○	◐	1	○	○	0
Tiny Flashlight	○	○	0	○	○	0	○	○	0	○	○	0
Celebrity Care	●	●	5	○	◐	0	○	◐	0	○	○	0
Mako Mobile	○	○	1	○	○	1	○	○	0	○	○	0
Video Poker	○	○	2	○	○	2	○	○	0	○	○	0
Check: Bill & Money	○	○	3	○	○	2	○	○	2	○	○	0
Princess Nail Salon	●	●	5	○	◐	0	○	◐	0	○	○	0
Extreme Droid Jump	○	○	0	○	○	0	○	○	0	○	○	0
Transparent Screen	●	●	3	●	●	3	●	●	3	○	○	0

Crash: ● Application crashes ○ Application loads successfully
Side Effects: ● Major side effects ◐ Minor side effects ○ No side effects

Table 2: Crashes, visual side effects, and run-time errors observed when running the applications using the four configurations.

Coarse Tracking Object-level tracking, labeled strings are replaced with arbitrary ones (even if only partially labeled).

No constraints Byte-level tracking, labeled characters are replaced with arbitrary strings (without conforming to functionality constraints).

Generic constraints Byte-level tracking, labeled strings are replaced while satisfying the generic constraints (but may violate application-specific constraints).

All functionality constraints Byte-level tracking, labeled strings are replaced while satisfying both generic and application-specific constraints.

A summary of which FASE features are used by each configuration is given in Table 1. The last configuration uses the full set of available features in the FASE system. For this configuration, we inspected the traces for each application and wrote application constraints in the FASE DSL. We inspected the sensitive information flows observed when running the application using the other three configurations, and then examined their logs and run-time errors to derive application-specific functional constraints.

To asses the robustness of each configuration, we ran each application using our user scripts and noted (i) whether the application crashes or otherwise loads successfully; (ii) whether there are major, minor or no visual side effects; and (iii) the number of unique run-time errors. We measure (i) and (iii) by inspecting the trace for each application using the Android adb tool. We used the methodology of [18] to measure (ii): we capture screenshots between user command, automatically highlight visual differences, and manually classify discrepancies into major (an essential visual element is missing), minor (a nonessential visual element, such as an ad, is missing) and none (the screenshots are identical).

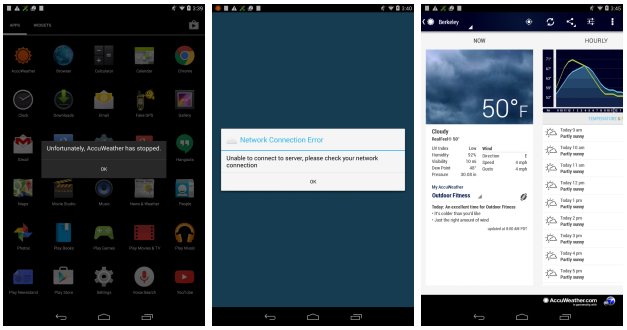
We present our results in Table 2. Using the *coarse* configuration, 40% of the applications crash, 50% show visual side effects, and 65% throw run-time errors. When fine-grained tracking is enabled, we observe fewer—but still a significant number of—problems. Remediation using generic constraints further reduces the observed side effects: 15% of the applications crash, 45% have visual side effects, and 50% throw run-time errors. Finally, using the *all functionality constraints* configuration only one application throws a run-time error while also exhibiting a minor visual side effect. Further analysis reveals, however, that this is not due to the FASE algorithm, the problem being an authentication error with the Google Maps API caused by resigning the application after instrumentation.

To gain better understanding of the results, we manually inspected the crashes and visual side effects. An example of an application that crashes due to violation of a generic constraint is Antsmasher, and we depict the relevant code fragment in Figure 5. The constructor argument, `s1`, is received from a JSON object, and if it is anonymized by replacing every character with `"x"`, then the `URL` constructor throws an exception, leaving the variable `url` uninitialized. Later, when the `doInBackground` method is called, the application throws a `NullPointerException` and crashes. Similar crashes also occur due to changes to unsanitized strings that represent file paths. All such paths must be sanitized to prevent path traversal attacks [23]. However, unconstrained modifications may result in invalid paths that then lead to exceptions thrown by the constructor.

AccuWeather is an application where respecting only the generic constraints is insufficient; see Figure 6. This appli-

```
1  public class DownloadManager
2    extends AsyncTask<String, Void, String> {
3    private URL url;
4    public DownloadManager(String s1,
5      String s2, AsyncTaskCompleteListener l) {
6      ...
7      try {
8        this.url = new URL(s1);
9        return;
10     } catch (MalformedURLException e) {
11       e.printStackTrace();
12     }
13   }
14
15   private String doInBackground(String[] s) {
16     ...
17     Connection c = this.url.openConnection();
18     // crash: null pointer exception
19     InputStream i = c.getInputStream();
20   }
21 }
```

Figure 5: The fragment from the Antsmasher application which shows that violating a generic constraint can crash the application.



(a) Coarse Tracking (b) Generic Constr. (c) All Constraints

Figure 6: The generic constraint alone is insufficient to avoid major side effects for the AccuWeather application.

cation transmits private data over the network as follows: `http://api.accuweather.com?apikey=..&lang=en&..` If the system modifies the argument `en` with an invalid language identifier such as `xx`, then the application's back-end server returns an error as it fails to recognize the language, as shown in Figure 6b. Note, however, that the modified URL string satisfies the generic sink constraint. To avoid this problem, we wrote an application constraint that allows the application to send this information to its back-end server, resulting in absence of any visual side effects, as Figure 6c shows.

To summarize, our results provide evidence that FASE's key features—fine-grained tracking, generic constraints, and application-specific constraints—are necessary and sufficient to secure applications while avoiding side effects. The FASE system therefore meets our robustness engineering requirement.

R2: Overhead. To check whether the FASE system meets our second engineering requirement, we measured the overhead incurred by the system.

To precisely measure the overhead, we measured the total CPU time each application needs to complete all user actions defined in our user scripts. We deliberately measure CPU time, instead of measuring the time to complete a task, to avoid the noise due to the scheduling of unrelated events. For our experiments we used a stock Android device alongside a modified Android device that features the FASE tracking engine. We ran each application 10 times.

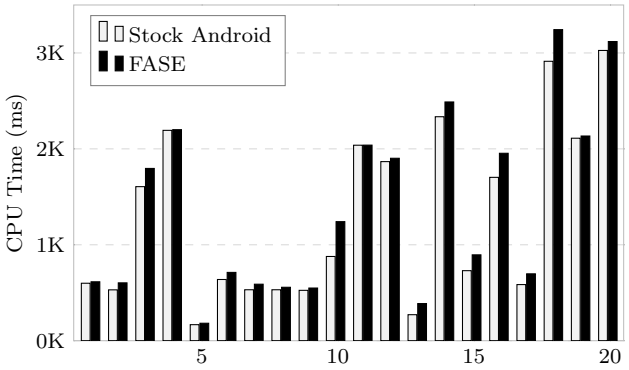


Figure 7: CPU time for running the 20 applications, using our user scripts, on a stock Android and a FASE device.

We plot the results in Figure 7. The data shows that the average overhead for real-world applications is 11.7%. Despite the fine granularity of the tracking engine, this overhead is in line with state-of-the-art run-time security systems (e.g. TaintDroid [10] and [31]). We remark that as mobile applications are event driven, this overhead is hardly felt in practice.

R3: Constraint Conciseness. This requirement reflects the usability of FASE. If the application constraints are large and complex, then it may be hard for developers to write them.

First, note that the generic constraints are already defined and are packaged into the FASE system. The developer should therefore only write application-specific constraints.

Second, we have observed that for most applications a single application constraint is sufficient to fix one run-time error. The average size of all application constraints among the selected applications is 4. We spent much less than an hour to write all constraints for an application. We therefore believe that a developer who is familiar with the FASE syntax can write the constraints in several minutes.

Most the constraints were indeed intuitive to write. To illustrate this, the constraint that prevents the run-time error in the AccuWeather application shown in Figure 6 is:

```
sink java.net.URL(url)
  if url.startsWith("api.accuweather.com") then
    keep url⟨Lang⟩
```

We did not encounter application constraints that have complex encoding in the FASE DSL.

Although developers may resort to more complex constraints to account for behaviors that we did not observe in our experiments, the average number of application constraints is likely to be in the same range (i.e. approximately 10 rules). This suggests that it is feasible to write application constraints for real-world applications in the FASE DSL.

7. RELATED WORK

There are numerous static and dynamic approaches for information-flow tracking. For static analyses, we refer the reader to [2, 12] and the references therein. Several approaches, such as [26], use static data flow trackers to augment applications with policy-enforcement capabilities. While these approaches often feature efficient enforcement, they cannot enforce complex, dynamic policies, such as *mask the last six digits of the IMSI before it is sent to an unknown server*, due to the inherent imprecision of their static analysis. As for the dynamic approaches, the state-of-the-art data flow tracking system for Android, TaintDroid [10], accounts for flows through variables and methods, as well as files and messages exchanged between applications. While TaintDroid also features low overhead, it supports neither fine-grained tracking nor run-time remediation.

Extensions of TaintDroid include AppFence [18], which (i) substitutes shadow data in place of confidential data and (ii) blocks network transmissions that contain private values; and Kynoid [29], which extends TaintDroid with user-defined security policies. These solutions are not sensitive to the target application's functionality, and inherit TaintDroid's coarse-grained tracking.

Several dynamic approaches enforce information-flow security by computing policy-compliant values based on explicit high- and low-views of the sensitive values. The Jeeves system [32], and its extension Jeeves* [5], enables developers to specify and enforce policies that describe which views of the sensitive values should be exposed to a sink. Approaches based on faceted values, such as [4], guarantee policy-compliant outputs by simulating program executions with different views of the sensitive values. While offering strong security guarantees, these systems have more complex tracking mechanisms: the Jeeves programming model tracks symbolic constraints and relies on SMT solving to enforce policies, while [4] simulates multiple executions. In contrast to these approaches, the FASE approach relies only on data flow tracking which can be efficiently implemented.

In the area of automatic remediation, Livshits and Chong propose a system for automated sanitizer placement [19]. To keep the run-time overhead low, the sanitizers are placed statically whenever possible, and dynamically otherwise. Similarly, ScriptGard [28] dynamically corrects instances of misplaced sanitization. The problem of sanitizer placement does not address the challenge of correctly anonymizing/sanitizing data [30]. FASE addresses this issue with sensitivity to application functionality, complementing existing work on correct placement of endorsement functions.

BEK [15] is an expressive DSL for encoding sanitizers, which are traditionally hard to get right. The language is amenable to precise analysis for idempotence, commutativity, and equivalence properties. Unlike BEK, which developers use to specify how sensitive data is sanitized, the FASE DSL expresses *functionality constraints*, which formalize requirements made by the program that security enforcement must respect.

Several works have presented character-level dynamic taint tracking for Web applications [8, 13, 33] with comparable overhead to the FASE tracking engine, which is also due to instrumenting only the string library. Unlike our FASE system, these systems do not account for primitive values, and defend only against integrity threats. Also, run-time remediation is limited to blocking operations without rewriting values with sensitivity to functionality.

The FASE tracking engine for primitive values is similar to BayesDroid [31], a dynamic system for detecting confidentiality leaks using Bayesian reasoning. BayesDroid, however, is designed specifically for privacy analysis; it does not perform online remediation.

Finally, for the solver component, there are existing general-purpose solvers for string constraints [34, 11, 14, 16, 6]. These decide standard regular and CFL problems, including language membership, intersection and equivalence. Such solvers can compute the intersection between FASE's regular application constraints and context-free generic constraints (step 5 of Algorithm 1). The FASE solver extends the solver of the ACLA framework [6] with support for deriving regular expressions from application constraints expressed in our DSL and (partially) labeled strings.

8. CONCLUSION AND FUTURE WORK

We presented functionality-aware security enforcement, a lightweight approach for online information-flow enforcement without disrupting the functionality of applications. FASE's key components are: (i) application and generic constraints, which capture the intrinsic functional needs of the applications and their libraries, (ii) a byte-level data flow engine tracking sensitive value at run time, and (iii) an run-time synthesizer for repairing sensitive values using constraint-compliant ones.

We presented a FASE implementation for Android and reported on experiments over popular Android apps. Our results show that the FASE system incurs an overhead of roughly 10% and achieves side-effect-free enforcement.

In the future, we plan to enhance FASE with inference capabilities, allowing synthesis of functionality constraints from static analysis of the app and/or dynamic (sandboxed) monitoring of the app's execution. We also plan to expand the concept of functionality-aware security enforcement in further dimensions beyond information flow, such as access control.

9. ACKNOWLEDGMENTS

We thank our shepherd Nick Nikiforakis and the anonymous reviewers for their valuable feedback. Some of the icons used in this work are designed by Freepik from Flaticon.

10. REFERENCES

[1] XPrivacy. www.xprivacy.eu/.
[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, 2014.
[3] http://asm.ow2.org.
[4] T. H. Austin and C. Flanagan. Multiple Facets for Dynamic Information Flow. In *POPL*, 2012.
[5] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted Execution of Policy-agnostic Programs. In *PLAS*, 2013.
[6] C. Brabrand, R. Giegerich, and A. Møller. Analyzing Ambiguity of Context-free Grammars. In *CIAA*, 2007.
[7] http://www.brics.dk/grammar/.

[8] E. Chin and D. Wagner. Efficient Character-level Taint Tracking for Java. In *SWS*, 2009.
[9] https://code.google.com/p/dex2jar/.
[10] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Real-time Privacy Monitoring on Smartphones. In *OSDI*, 2010.
[11] V. Ganesh, A. Kieżun, S. Artzi, P. Guo, P. Hooimeijer, and M. Ernst. HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection. In *CAV*. 2011.
[12] M. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.
[13] W. G. J. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *FSE*, 2006.
[14] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, 1995.
[15] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security*, 2011.
[16] P. Hooimeijer and W. Weimer. A Decision Procedure for Subset Constraints over Regular Languages. In *PLDI*, 2009.
[17] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
[18] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *CCS*, 2011.
[19] B. Livshits and S. Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *POPL*, 2013.
[20] M. Mohri and M. jan Nederhof. Regular Approximation Of Context-Free Grammars Through Transformation, 2000.
[21] M.-J. Nederhof. Practical Experiments with Regular Approximation of Context-free Languages. *Computational Linguistics*, 2000.
[22] OWASP Web Application Security Project, https://www.owasp.org/.
[23] OWASP. Path Traversal Attack. https://www.owasp.org/index.php/Path_Traversal.
[24] OWASP Mobile Security Project, https://www.owasp.org/index.php/OWASP_Mobile_Security_Project.
[25] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS*, 2014.
[26] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. In *ARES*, 2014.
[27] Uniform Resource Identifiers (URI): Generic Syntax, http://www.ietf.org/rfc/rfc2396.txt.
[28] P. Saxena, D. Molnar, and B. Livshits. ScriptGard: Automatic Context-sensitive Sanitization for Large-Scale Legacy Web Applications. In *CCS*, 2011.
[29] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: Real-Time Enforcement of Fine-grained, User-defined, and Data-centric Security Policies for Android. In *WISTP*, 2012.
[30] T. Tateishi, M. Pistoia, and O. Tripp. Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic. In *ISSTA*, 2011.
[31] O. Tripp and J. Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *USENIX Security*, 2014.
[32] J. Yang, K. Yessenov, and A. Solar-Lezama. A Language for Automatically Enforcing Privacy Policies. In *POPL*, 2012.
[33] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. In *SOSP*, 2009.
[34] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *FSE*, 2013.

APPENDIX

A. TEST SUBJECTS DETAILS

In Figure 8 we give statistics about the selected apps. For each app, we list its number of source and sink APIs, source and sink call sites, as well as detected sensitive information-flows.

B. FORMAL SEMANTICS OF THE FASE DSL

Here we formalize the FASE DSL semantics. We assume a standard *environment* $\Delta \in \mathit{Env} : \mathit{Vars} \to \mathit{Objs} \cup \mathit{Prims}$ mapping variables to objects (including strings) and primitive values. For readability we do not model the heap, and we assume that field identifiers are dereferenced in the standard way. The *labeling* $\tau \in \mathit{Lab} : \mathit{Objs} \cup \mathit{Prims} \to \mathcal{P}(\mathit{Labels})$ maps objects and primitives to their labels, as described in Section 3.2. We designate the label $l_\perp \in \mathit{Labels}$ to represents public/trusted data. We overload τ and write $\tau(x)$, where $x \in \mathit{Vars}$ is a variable, for $\tau(\Delta(x))$, i.e. $\tau(x)$ returns the label assigned to the object/primitive value stored at x. A *state* $\sigma = (\Delta, \tau) \in \mathit{Env} \times \mathit{Lab}$ defines the current environment and labeling function.

An application constraint maps a sink *snk*, a state σ, and a variable x to a set of strings encoded as a regular expression r. We define the derivation of r below.

Given a state σ and a string v, the relation $\vdash$ defines the grounding of symbolic regular expressions r_{sym}, i.e. regular expressions that contain variables and the keyword **val**, to concrete regular expressions r_{conc}; see Figure 10. The keyword **val** evaluates to the string v, constant strings s evaluate to s, and variables x evaluate to $\Delta(x)$. Composite regular expressions are evaluated recursively.

The satisfaction relation $\models$ given in Figure 9 formalizes the evaluation of conditions for a given state σ.

Given a state σ, a variable x, and a label l, we define $\mathit{block}(\sigma, x, l) \subseteq \mathbb{N} \times \mathbb{N}$ as:

$$(i,j) \in \mathit{block}(\sigma, x, l) \\ \Updownarrow \\ \tau(x[i,j]) = l \wedge \tau(x[i-1]) \neq l \wedge \tau(x[j+1]) \neq l \ .$$

A pair (i, j) is contained in $\mathit{block}(\sigma, x, l)$ iff the substring $x[i, j]$ is a substring block uniformly labeled with l.

App Name	Package Name	Source APIs	Sink APIs	Sources	Sinks	Sensitive Flows
Candy Crush Saga	com.king.candycrushsaga	49	64	250	956	3
Yellow Pages	com.avantar.wny	103	95	975	1647	13
Paper Toss	com.bfs.papertoss	33	54	128	579	10
Smiley Pops	com.boolbalabs.smileypops	55	55	578	667	5
Coffee Finder	com.brennasoft.findastarbucks	80	87	564	1505	6
Bump	com.bumptech.bumpga	51	67	329	708	1
iHeartRadio	com.clearchannel.iheartradio.controller2	27	51	117	235	7
SmartTacToe	com.dynamix.mobile.SmartTacToe	31	36	85	160	5
AccuWeather	com.accuweather	89	87	1208	1683	18
Antsmasher	com.bestcoolfungames.antsmasher	64	72	441	1060	18
aiMinesweeper	artfulbits.aiMinesweeper	24	44	110	550	1
Cat Hair Saloon	com.coolfish.cathairsalon	40	60	202	602	10
Tiny Flashlight	com.devuni.flashlight	44	70	360	499	2
Celebrity Care	com.g6677.android.cbaby	44	54	279	594	11
Mako Mobile	com.goldtouch.mako	91	95	973	2236	23
Video Poker	com.infimosoft.videopoker	81	80	898	1545	39
Check: Bill Pay	com.netgate	80	92	948	2194	23
Princess Nail Salon	com.g6677.android.pnailspa.apk	44	54	279	594	15
Extreme Droid Jump	com.electricsheep.edj	49	61	316	906	2
Transparent Screen	com.digisoft.TransparentScreen	61	71	286	1139	14

Figure 8: Information about the Android apps used in our experiments.

$$\boxed{\sigma \models \mathit{Cond}}$$

$$\frac{\mathit{bool\_op}(\Delta(x_1),\ldots,\Delta(x_k)) \neq \mathsf{false}}{(\Delta,\tau) \models \mathit{bool\_op}(x_1,\ldots,x_k)}\ \textsc{BOp}$$

$$\frac{\sigma \not\models c}{\sigma \models (\textbf{not}\ c)}\ \textsc{Not} \qquad \frac{\sigma \models c_1 \quad \sigma \models c_2}{(\sigma \models\ c_1\ \textbf{and}\ c_2)}\ \textsc{And}$$

Figure 9: Satisfaction relation between states and conditions

Let $R^\star$ denote the regular expression that accepts all strings. We write $(x\langle l\rangle\ \textbf{to}\ r)$ for $(\textbf{constrain}\ x\langle l\rangle\ \textbf{to}\ r)$ to avoid clutter. Given an expression $(x\langle l\rangle\ \textbf{to}\ r)$, a state $\sigma = (\Delta,\tau)$, a variable x, and $p, q \in \mathbb{N}$, we define α as

$$\begin{array}{ll}
\alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x[p,q]\big) = x[p,q] & \text{if } \tau(x[p,q]) = l_\perp \\
\alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x[p,q]\big) = R^\star & \text{if } \tau(x[p,q]) = l' \neq l \\
\alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x[p,q]\big) = r_1 \cdot r_2 \cdot r_3 & \text{otherwise}
\end{array}$$

where for some $(i,j) \in \mathit{block}(\sigma, x, l)$, we have

$$\begin{array}{l}
r_1 = \alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x[p,i]\big)\ , \\
\sigma, x[i,j] \vdash r \Downarrow r_2, \text{ and} \\
r_3 = \alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x[j,q]\big)\ .
\end{array}$$

Here α evaluates a substring $x[p,q]$ to itself if it is not labeled, to the accept-all regular expression $R^\star$ if $x[p,q]$ is a block assigned with a label other than the one in the constraint, and otherwise it recursively evaluates its labeled blocks. We write $\alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x\big)$ as a shorthand for for $\alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x[0, \mathit{len}(x)]\big)$.

The semantics of a set A of constraints is defined as:

$$[\![A]\!](\sigma, \mathit{snk}, x) = \bigcap \big\{\alpha\big((x\langle l\rangle\ \textbf{to}\ r), \sigma, x\big) \mid (\mathit{snk}\ \textbf{if}\ c\ x\langle l\rangle\ \textbf{to}\ r) \in A \wedge \sigma \models c\big\}\ .$$

$[\![A]\!]$ returns the intersection of all applicable constraints. We remark that our synthesizer translates the regular expressions into DFAs, and then uses the standard algorithm for intersecting DFAs; see Section 5.

$$\boxed{\sigma, v \vdash r_{\mathsf{sym}} \Downarrow r_{\mathsf{conc}}}$$

$$\frac{}{\sigma, v \vdash s \Downarrow s}\ \textsc{Str} \qquad \frac{}{(\Delta,\tau), v \vdash x \Downarrow \Delta(x)}\ \textsc{Var}$$

$$\frac{s = \mathit{str\_op}(\Delta(x_1),\ldots,\Delta(x_n))}{(\Delta,\tau), v \vdash \mathit{str\_op}(x_1,\ldots,x_n) \Downarrow s}\ \textsc{StrOp}$$

$$\frac{}{\sigma, v \vdash \textbf{val} \Downarrow v}\ \textsc{Val} \qquad \frac{\sigma, v \vdash r \Downarrow r_c}{\sigma, v \vdash r^\star \Downarrow (r_c)^\star}\ \textsc{Star}$$

$$\frac{\sigma, v \vdash r \Downarrow r_c \quad \sigma, v \vdash r' \Downarrow r'_c}{\sigma, v \vdash (r \cdot r') \Downarrow (r_c \cdot r'_c)}\ \textsc{Conc}$$

$$\frac{\sigma, v \vdash r \Downarrow r_c \quad \sigma, v \vdash r' \Downarrow r'_c}{\sigma, v \vdash (r + r') \Downarrow (r_c + r'_c)}\ \textsc{Alt}$$

Figure 10: Grounding symbolic regular expressions