Towards a Taint Mode for Cloud Computing Web Applications

Luciano Bello
Chalmers University of Technology
bello@chalmers.se

Alejandro Russo
Chalmers University of Technology
russo@chalmers.se

Abstract

Cloud computing is generally understood as the distribution of data and computations over the Internet. Over the past years, there has been a steep increase in web sites using this technology. Unfortunately, those web sites are not exempted from injection flaws and cross-site scripting, two of the most common security risks in web applications. Taint analysis is an automatic approach to detect vulnerabilities. Cloud computing platforms possess several features that, while facilitating the development of web applications, make it difficult to apply off-the-shelf taint analysis techniques. More specifically, several of the existing taint analysis techniques do not deal with persistent storage (e.g. object datastores), opaque objects (objects whose implementation cannot be accessed and thus tracking tainted data becomes a challenge), or a rich set of security policies (e.g. forcing a specific order of sanitizers to be applied). We propose a taint analysis for could computing web applications that consider these aspects. Rather than modifying interpreters or compilers, we provide taint analysis via a Python library for the cloud computing platform Google App Engine (GAE). To evaluate the use of our library, we harden an existing GAE web application against cross-site scripting attacks.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: Protection Mechanisms

General Terms Security, Languages

Keywords Taint analysis, Cloud computing, Web applications, Library, Python

1. Introduction

Cloud computing is a model to enable ubiquitous, convenient, and on-demand network access to some computing resources [31]. Due to its cost-benefit ratio, on-demand scalability and simplicity, cloud computing is spreading quickly among companies. By paying a (relatively) small fee, companies are relieved from big investments on servers, database administrators and backup systems to run their web sites. Cloud computing developers often use a platform that provides facilities to access persistent storage as if it were a local resource. In this manner, it is easy to dynamically accommodate or change in which part of the cloud computing infrastructure the application gets executed.

Recent studies show that attacks against web applications constitute more than 60% of the total attempts to exploit vulnerabilities online [36]. Web sites running in the cloud are not exempted from this. When development of web applications is done with little or no security in mind, the presence of security holes increases dramatically. Web-based vulnerabilities have already outpaced those of all other platforms [11] and there is no reason to believe that this tendency is going to change soon [19]. OWASP's top ten security risks has established injection flaws and cross-site scripting as the most common vulnerabilities in web applications [1, 40]. Although these attacks are classified differently, they are produced by the same reason: *user input data is sent to a sensitive sink without a proper sanitization*. For instance, injection flaws could occur when user data is sent to an interpreter as part of a system call executing unintended commands. To harden applications against these attacks, popular web scripting languages provide taint analysis [2, 3].

Taint analysis is an automatic approach to spot potential vulnerabilities. Intuitively, taint analysis restricts how tainted (untrustworthy) data flows inside programs, i.e., it constrains data to be untainted (trustworthy) or previously sanitized when reaching sensitive sinks. The analysis is then able to detect simple programming errors like forgetting to escape some characters in a string when building HTML web pages. It is worth mentioning that taint analysis is not conceived for scenarios where the attacker has control over the code.

Taint analysis comes in different forms and shapes. The analysis can be perform statically [23, 28, 39], dynamically [2, 3, 20, 21, 25, 34, 38, 44], or both [12, 14, 22, 29, 41, 45]. Traditionally, taint analysis tends to only consider strings or characters [2, 20, 21, 25, 33, 38] while ignoring other data structures or built-in values. The analysis can be provided as a special mode of the interpreter called *taint mode* (e.g. [2],[3],[33],[25]) or via a library [15]. Enhancing an inter-

PLAS'12 June 15, Beijing, China.
Copyright © 2012 ACM ISBN 978-1-4503-1441-1/12/06. . . $10.00

preter with taint mode generally requires to carefully modify its underlying data structures, which is a major task on its own. In contrast, Conti and Russo[15] show how to use some programming languages abstraction provided by Python in order to provide taint mode as a library. By staying at the programming language level, the authors show that the taint mode can be easily adapted to consider a wider set of built-in types (e.g. strings, integers, and unicode). Changing the source code of a library is a much easier task than changing an interpreter.

Google App Engine (GAE), a popular platform for developing and hosting web applications in the cloud, does not provide automatic tools to help developers avoid injection flaws or cross-site scripting (XSS) vulnerabilities. GAE possesses several features that, while facilitating programming, make the application of off-the-shelf taint analysis techniques difficult. Although we focus on GAE, similar difficulties arise when trying to apply taint analysis to other could computing development platforms [1]. More specifically, we identify the following aspects.

- **Persistent storage**: To the best of our knowledge, the taint analysis described in [18] is the only one considering persistent storage. Generally speaking, taint analysis avoids keeping track of taint information in datastores by, for instance, forcing data sanitization before it is committed. While this seems to be a reasonable strategy, it might be inadequate for most web applications. When users post entries into a forum, it is a common practice to store a copy of their original, unmodified submission so that changes to the way data is sanitized (or formatted) can be easily applied to older submissions.
- **Opaque objects**: To boost performance, the GAE platform includes some customized libraries. The interface of these libraries are often presented as opaque objects, i.e., objects which internal structure cannot be accessed from the web application. Opaque objects are usually a mechanism to restrictively allow calling code written in C (or any other high-performance language) from the GAE platform. Since the internal structure of such objects is not visible from the interpreter, it is difficult to perform tracking of tainted information, i.e. it is difficult to determine how the output of a given method depends on the (possibly tainted) input arguments.
- **Sanitization policies**: Web frameworks provide some standard sanitization functions that can be composed to create more complex sanitizers. It is common that web frameworks do not enforce the correct use of sanitizers [43]. For instance, applications might require that some data is sanitized using several sanitizers in any, or a specific, order. Traditionally, taint analysis does not support such fine-grained policies [2, 12, 20, 21, 23, 25, 28, 39, 45].

[1] Amazon AWS https://aws.amazon.com/articles/3998, Windows Azure http://www.windowsazure.com/en-us/home/scenarios/web/

In this work, we present a Python library that provides taint analysis for web applications written in GAE for Python. The implemented taint mode library propagates taint information as usual (i.e. data derived from tainted data is also tainted) while considering persistent storage, opaque objects and a rich set of security policies. Users of the library can run the taint mode by performing minimal modifications to applications' source code. The library uses security lattices [16] as the interface to express a rich set of sanitization policies. Surprisingly, we find that the least upper bound operation ($\sqcup$) is often not suitable to capture the security level of aggregated data when sanitizers lack compositionality properties. In fact, popular web frameworks often provide such problematic sanitizers. Instead of $\sqcup$, we introduce the operator $\curlyvee$ that computes upper bounds (not necessarily the least ones). To evaluate and motivate the use of our library, we harden the implementation of an existing web application written using GAE for Python. The library and the modified example can be downloaded at `http://www.cse.chalmers.se/~russo/GAEtaintmode/`.

1.1 Motivating example

The google-app-engine-samples project [7] stores examples of simple web applications for GAE. The *guestbook* application [8] used by the *Getting Started* documentation [6] serves as the running example along the paper. Although this application is rather simple, it contains all the ingredients to show how our taint mode works. The guestbook application consists of a web page where anonymous or authenticated users are able to write greeting messages which are stored into the GAE object datastore. These messages are fetched every time a user visits the web page. This application involves user inputs (greeting messages), the GAE object datastore, and the presence of opaque objects (due to the use of a web framework to build HTTP responses). When building the main web page, the application executes the following lines of code for every message being fetched from the object datastore:

```
<blockquote>{{ greeting.content|escape
  }}</blockquote>
```

The variable `greeting.content` is replaced by a greeting text and is subsequently fed to the sanitizer `escape` which replaces characters that might produce injection attacks such as < and >. We show that, by using our library, the taint analysis raises an alarm if the programmer omits to apply the sanitizer `escape` to every greeting message. Moreover, the library is able to enforce a specific sanitization policy, e.g., that greeting messages should be cleaned by applying some specific sanitizers in a specific order.

The paper is organized as follows. Section 2 gives background information on taint analysis. Section 3 describes how taint information gets propagated into the object datastore. Section 4 deals with taint analysis for opaque objects. Section 5 illustrates different security policies supported by our taint analysis. Section 6 incorporates taint analysis to our motivating example. Section 7 presents related work. Conclusions and future work are stated in Section 8.

2. Taint analysis

Taint analysis keeps track of how user inputs, or tainted data, propagate inside programs by focusing on assignments. Intuitively, when the right-hand side of an assignment uses a tainted value, the variable appearing on the left-hand side becomes tainted. Taint analysis can be seen as an information-flow mechanism for integrity [13]. In fact, taint analysis is nothing more than a tracking mechanism for explicit flows [17], i.e., direct flows of information from one variable to another. Implicit flows, or flows through the control-flow constructs of the programming language, are usually ignored. The following piece of code shows an implicit flow.

```
if tainted == 'a' : untainted = 'a'
else : untainted = ''
```

Variables `tainted` and `untainted` are initially tainted and untainted, respectively. The taint analysis determines that after executing the branch, variable `untainted` remains untainted since it is assigned to untainted constants on both branches, i.e., the strings `'a'` and `''`. Yet, the value of `tainted` is copied into `untainted` when `tainted == 'a'`! If attackers have full control over the code (i.e. attackers can write the code to be executed), taint analysis is easily circumvented by implicit flows. There is a large body of literature on language-based information-flow security regarding how to track implicit flows [35]. There are scenarios, however, where taint analysis is helpful, e.g., non-malicious applications. In such scenarios, the attacker's goal consists of exploiting vulnerabilities by providing crafted input. It is then enough that programmers simply forget to call some sanitization function for a vulnerability to be exposed.

It is unusual to find formalizations that capture semantically, and precisely, what security condition taint analysis enforces. To the best of our knowledge, the closest formal semantic definition is given by Volpano [42]. Nevertheless, Volpano's definition cannot be fully applied to taint analysis because it ignores sanitization (or endorsement) of data. To remedy that, it could be possible to extend Volpano's definition using intransitive noninterference, but this topic is beyond the scope of this paper. It is not so easy to make a precise and fair appreciation of the soundness and completeness of a given taint analysis technique. On one hand, completeness is a challenging property for any kind of analysis. We do not expect taint analysis to be an exception to that. On the other hand, assuming the policy *untrustworthy data should not reach sensitive sinks*, some taint analysis may be unsound due to implementation details. For instance, analyses focusing only on strings do not propagate taint information when the right-hand side of the assignment is an integer (e.g. [2, 20, 21, 25, 29, 33, 34]). The following piece of code shows how to encode a tainted character as an untainted integer.

```
untainted_int = ord(tainted_char)
```

Variable `untainted_int` can be casted back into a string and be used into a sensitive sink without being sanitized! Regardless of this point, taint analysis has been successfully used to prevent a wide range of attacks like buffer overruns (e.g. [24, 32]), format strings (e.g. [14]), and command injections (e.g. [12, 28]). The practical value of taint analysis is given by how easy it can be applied and how often it captures omissions with respect to sanitization of data.

2.1 Taint mode via a library

Rather than modifying interpreters, Conti and Russo provide a taint mode for Python built-in types via a library [15]. It is worth mentioning that built-in types are immutable objects in Python. The authors show how Python's object-oriented features and dynamic typing mechanisms can be used to propagate taint information. The core part of the library defines subclasses of built-in types. These subclasses, called *taint-aware* classes, contain the attribute `taints` used to store taint information. Methods of *taint-aware* classes are intentionally defined to propagate taint information from the input arguments, and the object calling those methods, into the return values. For instance, consider the following interaction with the Python interpreter.

```
1 > ts = taint('tainted string')
2 > ts.taints
3 True
4 > us = 'string'
5 > tainted(ts + us)
6 True
```

Function `taint` takes a built-in value and returns a taint-aware version of it. More specifically, Line 1 takes a string, i.e., an instance of the class `str`, and returns a tainted string. Tainted strings are instances of the class `STR`, which is a subclass of `str`. For simplicity reasons, we assume that the `taints` attributes are simply boolean variables. However, it can be as complex as any metadata related to taints (see Section 5). Line 2 shows the `taints` attribute of `ts`. Line 4 declares an untainted string `us`. Function `tainted` returns a boolean value indicating if the argument is tainted. Line 5 shows that the concatenation of a tainted string with an untainted one (`ts + us`) results in a tainted value. This effect occurs since `ts + us` gets translated into the invocation of the concatenation method of the most specific class, i.e., `STR`. Therefore, `ts + us` is equivalent to `ts.__add__(us)`, which propagates the taints from the object calling the method (`ts`) and the argument (`us`) into the result. Consequently, and as shown by this example, operators for different built-in types can be instrumented to propagate taint information by simply defining subclasses. This feature, and the fact that built-in operations are translated into object calls, is what makes Python particularly suitable to provide taint analysis via a library. It is difficult to applying these ideas to languages like PHP or ASP where strings are not objects and there are no obvious mechanisms to instrument string operations without modifications in the underlying runtime system [20, 29, 33]. In contrast, the programming language Ruby provides some mechanisms to instrument operations [2] that could be possibly used to provided taint analysis via a library. Based on the ideas of Conti and Russo, we develop a taint mode that goes beyond built-in values.

3. Persistent storage

Taint analysis often does not propagate taint information into persistent storage such as files or databases. Instead, data must be sanitized before being saved. In the context of web applications, this strategy might be inadequate, e.g., it is often recommendable that web applications store user's data exactly as submitted to the web site. In that manner, changes in the way that information is formatted or sanitized can be applied to older submissions. In this section, we extend the GAE platform to support tainted values in the GAE object datastore.

The GAE object datastore saves (and retrieves) data objects in (from) the cloud computing infrastructure. These objects are called *entities* and their attributes are referred to as *properties*. Properties represent different types of data (integers, floating-point numbers, strings, etc). It is important to remark that a property cannot be an entity itself (and thus there is no need to consider a notion of nested tainting). An application only has access to entities that it has created itself. The GAE platform includes an API to model entities as instances of the class `db.Model`. For example, the guestbook application models messages by instances of the class `Greeting`.

```
class Greeting(db.Model):
  author = db.UserProperty()
  content = db.StringProperty(multiline=
  True)
  date = db.DateTimeProperty(auto_now_add=
  True)
```

A `Greeting` entity contains information about who wrote the entry (property `author`), its content (property `content`), and when it was written (property `date`). When a user writes a comment into the guestbook, the application creates an entity, fetches the comment from the HTML form field `content`, and saves it into the database. The following piece of code reflects that procedure.

```
greeting = Greeting()
greeting.author = users.get_current_user()
greeting.content = self.request.get('
  content')
greeting.put()
```

[2] http://stackoverflow.com/questions/1283977/existence-of-right-addition-multiplication-in-ruby

In this piece of code, the object `self` refers to a handler given to the application to access the fields submitted by the POST request. Method `greeting.put()` saves the entity into the datastore. The GAE platform provides two interfaces to fetch entities from the datastore: a query object interface, and a very simplified SQL-like query language called Google Query Language (GQL). Due to lack of space, we only show how the library works for the query object interface. This interface requires to create a query object by, for example, calling the method `all` of an entity model. Using that object, the guestbook application is able to fetch all the greetings from the datastore. The following piece of code shows the use of `all`.

```
query = Greeting.all().order('-date')
greetings = query.fetch()
```

The first line creates a query object in order to select the `Greeting` entities ordered by date. The second line retrieves the entities from the datastore.

One of the main problems to add taint information to the GAE datastore is related to properties. GAE forces a typed discipline on the properties, i.e., it only allows properties to be of a specific type (e.g. string or integers) at the time of saving them into the datastore. This design decision makes impossible to simply store tainted values directly into the datastore. After all, a tainted string is not a built-in type but rather a value from a subclass of strings (recall Section 2.1). To overcome this problem, we implement a mechanism to extend entity models in order to account for taint information in a separate property.

Decorators in Python allow to dynamically alter the behavior of functions, methods or classes without having to change the source code. The library provides the decorator `taintModel` to indicate which entities keep track of taint information. For the guestbook application is enough to add the line `@taintModel` before the declaration of `Greetings`.

```
@taintModel
class Greeting(db.Model):
  ...
```

We use ... to represent the rest of the declaration of the class `Greeting`. Decorator `taintModel` is a function that receives and constructs a class. More specifically, the previous code can be considered equivalent to

```
class Greeting(db.Model):
  ...

Greeting = taintModel(Greeting)
```

Consequently, when referring to `Greeting` in the rest of the source code, it is referring to the class being returned by `taintModel`. This decorator extends the definition of `Greeting` by adding a new text property that stores a mapping from property names into taint information. The decorator also redefines the methods `put` and `fetch` to consider such mappings. When an entity gets saved into the datastore, the put method checks for tainted values and then builds a mapping reflecting the taint information in the attributes. We refer to this mapping as *tainting mapping*. After that, the tainted attributes are casted into their corresponding untainted versions (e.g. properties of type STR are casted into str) so that the entity can be saved together with the constructed tainting mapping. When an entity is fetched from the datastore, the method fetch reads the content of the entity and creates tainted values for those properties that the tainting mapping indicates. To illustrate this point, let us consider the following example based on the guestbook model.

```
1  > greeting1=Greeting()
2  > greeting1.content=taint('<script>')
3  > greeting1.put()
4  > greeting2=Greeting()
5  > greeting2.content='Hello!'
6  > greeting2.put()
7  > query=Greeting.all().order('-date')
8  > greetings = query.fetch()
9  > greetings[0].content
10 '<script>'
11 > tainted(greetings[0].author)
12 False
13 > tainted(greetings[0].content)
14 True
15 > tainted(greetings[0].date)
16 False
17 > greetings[1].content
18 'Hello'
19 > tainted(greetings[1].author)
20 False
21 > tainted(greetings[1].content)
22 False
23 > tainted(greetings[1].date)
24 False
```

Assuming an initially empty object datastore, lines 1–6 create and store two entities, where one of them contains the tainted string '<script>'. Lines 7–8 recover the entities from the datastore. Lines 9–16 show that only the property content is tainted for the first entity. Lines 17–24 show that the second entity has not tainted properties.

In principle, the user of the library needs to explicitly indicate (by using taintModel) what entities should propagate taint information into the datastore. Alternatively, it is also possible to extend the class db.Model to automatically support tainting mappings. By doing so, every entity class that inherits from db.Model is able to store taint information into the datastore.

4. Opaque objects

GAE applications involve objects. The taint mode by Conti and Russo [15] shows how to propagate taint information for built-in types. In Python, built-in values are treated as immutable objects. Conti and Russo's work does not consider mutable objects like user-defined objects. The fact that GAE includes some third-party libraries besides the standard library (with modifications) opens the door to opaque objects and forces the analysis to treat them differently than user-defined ones. In this section, we show how our library perform taint analysis for objects in their various flavors.

Our library does not have access to some attributes or implementation of some methods from opaque objects. This restriction makes it impossible to track taint information computed by such objects, e.g., it is not possible to determine how outputs depend on input arguments. To illustrate this point, we consider the well-known cStringIO module from the standard library. This module defines the class StringIO to implement objects representing string buffers. The implementation of this class is done in C and imported in Python, which introduces opaque instances of StringIO. Using the library by Conti and Russo, if we try to place a tainted value into a StringIO buffer, the taint information gets lost, i.e., the library cannot track how the tainted string is used by the opaque object. Let us consider the following piece of code.

```
1 > from cStringIO import StringIO
2 > buff=StringIO()
3 > buff.write(taint('<script>'))
4 > tainted(buff.getvalue())
5 False
```

Line 3 inserts a tainted string into the buffer. When reading the content of the buffer (Line 4), the resulting string is not tainted (Line 5).

We design our library to perform a coarse approximation related to taint information when dealing with opaque objects. More specifically, for every opaque object the library creates a wrapper object that contains taint information (the attribute taints) and wrapper methods to perform taint propagation accordingly. For any call to a method of an opaque object, the corresponding wrapper object propagates the taint information from the arguments of the method, and the object itself, into the returning value. The taint information of the opaque object gets then updated to the taint information of the resulting value.

By using the primitive opaque_taint, the user of the library indicates which are the opaque objects being used by the application. In principle, to avoid developer intervention, the library could automatically declare several objects provided with the GAE platform as opaque. The following code shows a possible use of opaque_taint.

```
1 > from cStringIO import StringIO
2 > buff=opaque_taint(StringIO())
3 > buff.write('us')
4 > tainted(buff)
5 False
6 > buff.write(taint('ts'))
7 > tainted(buff)
8 True
```

In Line 2, the `opaque_taint` primitive takes the opaque object and returns the corresponding wrapper object. Line 3–8 shows that the object is clean until a tainted argument is given. Consequently, obtaining the content of the buffer results in a tainted string as expected.

```
> tainted(buff.getvalue())
True
```

Since `buff` is tainted, the taint analysis determines that every value returned by any method call of that object is also tainted. Observe that we need to be conservative at this point since we do not know how outputs depend on inputs in opaque objects. It might happen that the analysis considers data as tainted even if that data is not related with the content of the buffer. For example, `StringIO` objects (as other classes that simulate a file object) has the attribute `softspace` used by the `print` statement to determine if a space should be add at the end of a printed string. Clearly, this attribute is not affected by reading or writing into the buffer; however, it gets tainted:

```
> tainted(buff.softspace)
True
```

The analysis looses precision at this point as the price to pay for not knowing the internal structure of the opaque object.

As any approximation, our approach to opaque objects might impact on permissiveness, e.g., it is possible to obtain tainted empty strings when the buffer of an `StringIO` object has run out of elements.

The library treats pure Python user-defined objects as merely containers, i.e., their attributes can be tainted. Therefore, whenever a tainted attribute is utilized for computing the result of a method call, the return value gets tainted. To illustrate the analysis of these objects, we consider the non-opaque version of `StringIO` provided by the module `StringIO`.

```
1 > from StringIO import StringIO
2 > buff=StringIO(taint('<script>'))
3 > hasattr(buff, 'buf')
4 True
5 > tainted(buff.buf)
6 True
7 > tainted(buff.getvalue())
8 True
```

Notice that it is possible to access to the attribute `buf` (line 3) which stores the string buffer, so the object `buff` is not opaque. Lines 5–6 show that the buffer is tainted. Lines 7–8 demonstrate that reading the content of the buffer results in a tainted string. Unlike the example for opaque objects, the attribute `softspace` does not necessarily is tainted although the buffer is.

```
> tainted(buff.softspace)
False
```

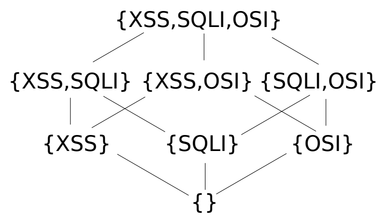

Figure 1. Order for set of tags identifying different vulnerabilities

5. Security policies

Inspired by information-flow research, some taint analyses ([14, 22]) use security lattices [16] to specify security policies. We use $\sqsubseteq$ to denote the order-relation of the lattice. Elements in the security lattice represent the integrity level of data. The bottom and top elements represent trustworthy and untrustworthy data, respectively. Lattices with more than two security levels allow for different degrees of integrity and thus expressing rich properties. With the security lattice in place, security levels are assigned to sources of user inputs and sensitive sinks. In general, user inputs are associated with the top element of the lattice (untrustworthy data). Sensitive sinks are often associated with security levels below top. Taint analysis allows data to flow into a sensitive sink provided that the integrity level of the data is equal or above the one associated with the sink. The higher the security level associated with a piece of data is, the more untrustworthy it becomes. The security level of aggregated data is determined as the least upper bound ($\sqcup$) of the security levels of the constitutive parts. Sanitization of data is the only action that moves data from higher positions in the lattice to lower ones, and thus making data more trustworthy. In the scenario of web applications, the use of $\sqcup$ might not be adequate due to sanitizers being often not compositional with respect to, for instance, string operations. We illustrate this point with concrete examples in the rest of the section. Instead of the $\sqcup$, we introduce the upper bound operator $\curlyvee$ to correctly determine the security level of aggregated data. For each sensitive sink at security level l_s, our analysis considers a set of security levels called the *safe zone*. Data associated with one of these levels can flow into the sensitive sink. Otherwise, the library raises an alarm. The *safe zone* is usually defined in terms of the $\sqsubseteq$-relation. We show three instances of security policies implemented by our library that capture the application of sanitizers in different manners.

5.1 Different kinds of sensitive sinks

The first example consists of encoding security policies able to categorize sensitive sinks. We use tags to represent that data might exploit different kinds of vulnerabilities. For this example, we assume tags `SQLI`, `XSS`, and `OSI` to indicate SQL injections, cross-site scripting, and operating system command injections, respectively. The elements of the lattice are sets of tags. Intuitively, a set indicates that data has not been sanitized for the vulnerabilities described by the tags.

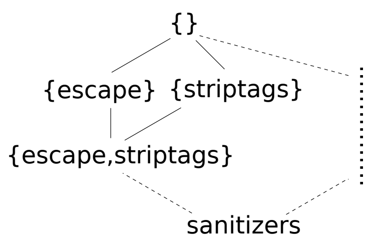

Figure 2. Order for identifying the application of sanitizers

The ⊑-relation is simply defined as set inclusion: $l_1 \sqsubseteq l_2$ iff $l_1 \subseteq l_2$. Figure 1 describes the order between the set of tags. User input is associated with the security level represented by the set of all the tags, in this case $\{\texttt{XSS}, \texttt{SQLI}, \texttt{OSI}\}$ representing fully untrustworthy data. The bottom element is the empty set denoting fully trustworthy data, i.e., data that is not part of any user input or has been sanitized to avoid every considered vulnerability. Sensitive sinks are then associated to possibly different security levels. For instance, the Python primitive `os.system`, which executes shell commands, can be associated to the security level $\{\texttt{OSI}\}$, while `db.select`, responsible to execute SQL-statement, can be associated to the security level $\{\texttt{SQLI}\}$. Given a sensitive sink at security level l_s, its safe zone is defined as every security level l such $\neg(l_s \sqsubseteq l)$. By doing so, tainted data with tags `SQLI` can, for instance, be used on sensitive sinks at security level `OSI` and vice versa. The upper bound operator $\curlyvee$ is given by set union. In this case, the upper bound operator coincides with the least upper bound induced by the ⊑-relation, i.e. $\curlyvee = \sqcup$. Sanitizers for a given tag t take data at security level l and endorsed it to the security level $l \smallsetminus \{t\}$, where $\smallsetminus$ is the symbol for set subtraction. Observe that $l \smallsetminus \{t\} \sqsubseteq l$.

5.2 Identifying the application of sanitizers

Different from the previous example, there are situations where developers want to know if data has been passed through specific sanitizers rather than knowing which vulnerability can be exploited. In this case the elements of the lattice are sets of sanitizers. A security level indicates which sanitizers have been applied to the data.

We define the order as follows: $l_1 \sqsubseteq l_2$ iff $l_2 \subseteq l_1$. Figure 2 illustrates an example for this order. The security level representing fully untrustworthy data is the empty set ($\{\}$), i.e., data that has not being applied to any sanitizer. User input is often associated to that level. On the other end, the set of all the existing sanitizers represents trustworthy data. We utilize the constant `sanitizers` to denote that set. Given a sensitive sink at security level l_s, its safe zone is defined as every security level l such $(l \sqsubseteq l_s)$. Therefore data flowing into a sensitive sink must have been applied to at least the sanitizers indicated by its security level.

Induced by ⊑, the least upper bound operator for this lattice is set intersection. However, this definition does not reflect the right security level for aggregated data. We define the upper bound $\curlyvee$ as a slightly more complicated operation than just intersection of sets. The reason for that relies in the compositional behavior of sanitizers. We say that a sanitizer is compositional if the result of sanitizing two pieces of data and then composing them is the same as firstly composing the pieces and then sanitizing. More specifically, we say that a sanitizer is compositional iff for all closed operations $\oplus$ and pieces of data d_1 and d_2, the following equation holds

$$\mathit{sanitizer}(d_1 \oplus d_2) = \mathit{sanitizer}(d_1) \oplus \mathit{sanitizer}(d_2).$$

Defining $\mathcal{N}$ as the set of non-compositional sanitizers, we define $l_1 \curlyvee l_2 = (l_1 \cap l_2) \smallsetminus \mathcal{N}$. In that manner, it is captured the fact that compositional sanitizers are only preserved when data gets combined. Observe that, in presence of non-compositional sanitizers, it holds that $(l_1 \sqcup l_2) \sqsubset (l_1 \curlyvee l_2)$.

```
> striptags('<b' + '> Be careful </b>')
' Be careful! '
> striptags('<b') + striptags('> Be
  careful </b>')
'<b> Be careful '
```

Figure 3. Non-compositionality of `striptags`

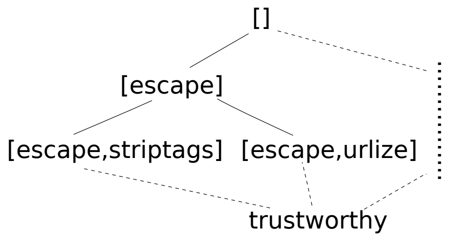

Figure 4. Order of application of sanitizers

To illustrate that non-compositional sanitizers exist in modern web frameworks, we consider two sanitizers from Django: `escape` and `striptags`. The sanitizer `escape` replaces some characters for their corresponding entity HTML name, e.g., character < is converted into the string `"&lt;"`. Since it only looks into one character at the time, `escape` is compositional. The sanitizer `striptags` removes HTML and XHTML tags from a given string, e.g., given the string `"<b> Be careful </b>"` results into the string `" Be careful "`. Observe that this sanitizer is non-compositional. To illustrate this point, consider the interaction with the Python interpreter given in Figure 3. Concatenating the strings *'<b'* and *'> Be careful </b>'* and then sanitizing is not the same as sanitizing *'<b'* and *'> Be careful </b>'* and then concatenating the results. According to the definition of $\curlyvee$, if we combine strings sanitized with `striptags`, the resulting string is associated with the security level $\{\}$, i.e., fully untrustworthy data ($\{\texttt{striptags}\} \curlyvee \{\texttt{striptags}\} = \{\}$). When a sanitizer t is applied to data at security level l, data is then endorsed to the security level $l \cup \{t\}$.

5.3 Applying sanitizers in a specific order

In some scenarios, it is important in which order sanitizers are applied. Different orders of application might lead to the presence of vulnerabilities. To illustrate this point, we consider the standard filter `urlize`: it detects if a string contains a URL and returns a clickable link if that is the case. For example:

```
> print urlize('www.chalmers.se')
<a href="http://www.chalmers.se" rel="
  nofollow">www.chalmers.se</a>
```

If the attacker is under control of the data being applied to `urlize`, it is possible to inject JavaScript code. An attacker can create a link that triggers JavaScript code when the mouse moves over it. More concretely, we have that

```
urlize('www."onmouseover="alert(42)"')
```

returns a tag element anchor that displays in the web browser the string `www."onmouseover="alert(42)"` and executes `javascript:alert(42)` when the mouse goes over it (no click needed):

```
<a href="http://www."onmouseover="alert
  (42)"" rel="nofollow">www."onmouseover="
  alert(42)"</a>
```

To avoid such injection attacks, it is recommended to first sanitize strings with `escape`.

```
urlize(escape('www."onmouseover="alert(42)
  "'))
```

In that manner, the resulting tag element anchor does not execute the JavaScript code.

```
<a href="http://www.&quot;onmouseover=&
  quot;alert(42)&quot;" rel="nofollow">www
  .&quot;onmouseover=&quot;alert(42)&quot
  ;</a>
```

We design a security lattice that accounts for the order in which sanitizers are applied. The elements of the lattice are lists of sanitizers. The order-relation is simply list prefix, i.e., $l_1 \sqsubseteq l_2$ iff l_1 is a prefix of l_2. Figure 4 illustrates partially a specific instance of this order. The empty list (`[]`) indicates that no sanitizer has been applied and therefore denotes untrustworthy data. User input is often associated to that level. In contrast, we introduce the constant `trustworthy` to encode any possible ordered application of sanitizers that provides fully trustworthy data. Given a sensitive sink at security level l_s, its safe zone is defined as every security level l such $(l \sqsubseteq l_s)$. Therefore, data flowing into a sensitive sink must have been applied to, at least, the sequence of sanitizers indicated at its security level.

Similar to the previous case, the least upper bound operator induced by $\sqsubseteq$ (i.e. the longest prefix) does not reflect the right security level of aggregated data. We use the term lists and security levels as interchangeable terms. We write $s : l$ to the list of sanitizers which first element is s and has a tail l. Taking into account the possibility of using non-compositional sanitizers ($\mathcal{N}$), the upper bound operator is defined as follows.

$$l_1 \curlyvee l_2 = \begin{cases} s : (l_1' \curlyvee l_2') & \text{, if } l_1 = s : l_1',\ \ l_2 = s : l_2',\ \ s \notin \mathcal{N} \\ [\,] & \text{, otherwise} \end{cases}$$

This definition essentially preserves the longest common prefix of compositional sanitizers, e.g., $[\texttt{escape}, \texttt{striptags}] \sqcup [\texttt{escape}]$ is $[\texttt{escape}]$. Observe that, in presence of non-compositional sanitizers, it holds that $(l_1 \sqcup l_2) \sqsubset (l_1 \curlyvee l_2)$. When applying a sanitizer s to data at security level l, the data is then endorsed to the security level $l +\!\!+ [s]$, where $+\!\!+$ denotes concatenation of lists.

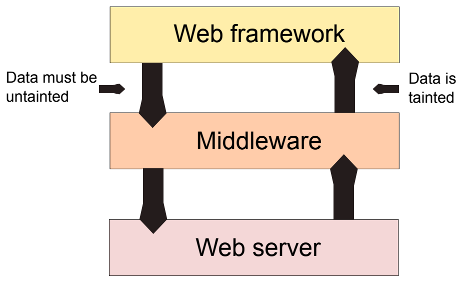

Figure 5. GAE platform schema

6. Hardening the motivating example

We revise the example from Section 1.1 and show how to adapt it to use our library. We have already described in Section 3 how to extend the entity `Greeting` so that taint information can be propagated to the datastore. In this section, we continue modifying the source code to indicate the sources of untrustworthy data, sensitive sinks, and sanitization functions. Since our library is specialized to the GAE platform, it allows us to provide out-of-the-box declaration for a set of operations that can be considered sources of tainted data as well as sensitive sinks. Sanitization functions, on the other hand, depend mainly on the web framework used for rendering web pages. If developers consider that the source of untrustworthy data, sensitive sink, or sanitizer are not precisely or correctly indicated, the library provides means to explicitly mark those operations in the code.

6.1 Source of tainted data and sensitive sinks in GAE

Our library considers the web server as both a source of tainted data and a sink sensitive to XSS attacks, i.e., data coming from the web server gets tainted while data going back to the server needs to be untainted. In order to understand how the library intermediates between the web server and the GAE framework, we need to shortly describe the GAE platform architecture.

GAE platform utilizes the concept of middleware, which is a standard way to intercept requests and responses between the web server and web frameworks (e.g. Django [5], CherryPy [4], Pylons [10], and webapp [6]). The GAE platform is, to a large extent, web framework independent, i.e., it runs any web framework that utilizes middlewares complying with the Web Server Gateway Interface (WSGI) [9]. Web frameworks are no more than a series of useful functions and a template systems to keep a separation between the presentation- (mostly written in HTML) and the logic part of the application. Figure 5 schematizes the interaction between the web server, middleware and web framework.

Our library provides its own middleware responsible to taint data coming from the web server (e.g. headers, strings send with the POST and GET methods, source IP and every other information from the client). The middleware also checks that data going back to the web server has been sanitized. As a sensitive sink, it is necessary to indicate the security level associated to the middleware. The library provides the configuration file `taintConfig.py` for that.

To extend the functionality of the guestbook application, we decide to allow users to include URL addresses in their greetings as long as they do not include XSS or other attacks. We then require that users' greetings must go through the sequence of sanitizers `escape`, `urlize`, and `shorturl` before reaching the web client. Sanitizers `escape` and `urlize` are described in Section 5. The user-defined sanitizer `shorturl` leaves URL inside an anchor tag only if they are short (e.g. like the ones provided by the Twitter link service). Clearly, the example demands the use of the security policy from Section 5.3 which considers the order of sanitizers. With this in mind, the file `taintConfig.py` looks as follows.

```
1  from Policies import *
2
3  policy = SanitizerOrders
4  policy.ssinks
5      = { 'middleware' :
6          ['escape','urlize','shorturl'] }
```

Line 3 indicates that the taint analysis takes into account the order in which sanitizers are applied. We define a mapping from sensitive sinks (`ssinks`) to security levels. In this case, Line 4–6 indicates that the middleware is associated with the security level represented by the list `['escape','urlize','shorturl']` and thus accepting only strings that have passed through the sequence of sanitizers `escape`, `urlize`, and `shorturl`. Once defined the configuration file, the library (called `taintmode`) should be imported by the application:

```
from ...
from taintmode import *
```

Since `taintmode` wraps some other modules' definitions, it is important to import it last. The motivating example, and WSGI applications in general, runs in the GAE's CGI environment by executing the following lines.

```
def main():
  run_wsgi_app(guestbook_app)
```

To use our customized WSGI middleware, those lines need to be slightly modified to simply include the procedure `TaintMiddleware` as follows.

```
def main():
  run_wsgi_app(TaintMiddleware(
  guestbook_app))
```

The guestbook application now gets tainted data from the web server and needs to sanitize data before sending it back to the web client.

6.2 Sanitization policies

The library needs to know which functions are sanitizers. It is also important to indicate if they are compositional. To do that, the file `taintConfig.py` needs to be extended as follows.

```
from Policies import *

policy = SanitizerOrders

policy.ssinks
    = { 'middleware' :
        ['escape','urlize','shorturl'] }

policy.sanitizers
    = { 'escape'   : Comp,
        'urlize'   : NonComp,
        'shorturl' : NonComp }
```

The variable `policy.sanitizers` defines a mapping from sanitizers' names to information required by the security policy implementation, i.e., `SanitizerOrders`. This information might change depending on the security policy to be enforced. For this scenario, we indicate whether a sanitizer is compositional (constant `Comp`) or non-compositional (constant `NonComp`).

If a developer forgets to apply `escape`, or applies the sanitizers in the wrong order, an exception (`TaintException`) is thrown indicating the tainted substring responsible for the alarm.

```
TaintException: wrong sequence of
  sanitizers ['urlize','shorturl']: ['<
  script>alert(42)</script>']
```

7. Related work

There is a large volume of published work describing taint analysis. Readers can refer to [14] for an excellent survey. In this section, we mainly refer to analyses developed for popular web scripting languages.

Perl [2] was the first scripting language to include taint analysis as a native feature of the interpreter. Different from our work, the security policy enforced by Perl's taint mode is rather static, i.e., strings originated from outside a program are tainted (e.g. inputs from users), sanitization is done by regular expressions, and files, shell commands and network connections are considered sensitive sinks. Ruby [3] provides a taint analysis similar to what our library does for opaque objects. However, our work allows for more precision in the analysis for non-opaque objects.

Several taint analysis have been developed for PHP. Aiming to avoid any user intervention, authors in [22] combine static and dynamic techniques to automatically repair vulnerabilities in PHP code. They propose to use a type-system to insert some predetermined sanitization functions when tainted values reach sensitive sinks. The semantics of programs might change when inserting sanitization functions, which constitutes the dynamic part of the analysis. We decide not to change the semantics of programs unless explicitly stated by the user of the library, i.e., we leave it up to the user of the library to decide where and how sanitization functions must be called. In [33], Nguyen-Toung et al. adapt the PHP interpreter to provide a dynamic taint analysis at the level of characters, which the authors call *precise tainting*. They argue that precise tainting gains precision over traditional taint analyses for strings. It would be interesting to see studies indicating how much precision (i.e. less false alarms) it is obtained with *precise tainting* in practice. Similarly to Nguyen-Toung et al.'s work, Futoransky [20] et al. provide a precise dynamic taint analysis for PHP. Pietraszek and Berghe [34] modify the PHP runtime environment to assign *metadata* to user-provided input as well as to provide metadata-preserving string operations. Security critical operations are also instrumented to evaluate, when taken strings as input, the risk of executing such operations based on the metadata. In our library, the attribute `taints` is general enough to encode Pietraszek and Berghe's metadata for strings. Jovanovic et al. [23] propose to combine a traditional data flow and alias analysis to increase the precision of their static taint analysis for PHP (which posses a 50% of false alarms rate). Different from our approach, Jovanovic et al. do not consider taints for objects. Focusing only on strings, the works in [12, 29] combine static and dynamic techniques. The static techniques are used to reduce the number of program variables where taint information must be tracked at runtime. The dynamic analysis in [12] consists of running test cases using attack strings rather than propagating taint information at runtime. Conversely, the work in [29] modifies the PHP virtual machine in order to propagate taint information. In particular, the modified virtual machine includes the field `labels` to store taint meta-information. This field is similar to the attribute `taints` used by our library.

A taint analysis for Java [21] instruments the class `java.lang.String` as well as classes that present untrustworthy sources and sensitive sinks. The authors mention that a custom class loader in the JVM is needed in order to perform online instrumentation. Another taint analysis for Java [39], called TAJ, focuses on scalability and performance requirements for industry-level applications. To achieve industrial demands, TAJ uses static techniques for pointer analysis, call-graph construction and slicing. Similar to our work, TAJ allows object fields to store tainted values (such objects are called *taint carries*). However, TAJ's static analysis does not consider persistent storage and opaque objects. The authors in [28] propose a static analysis for Java that focuses on achieving precision and scalability. Their analysis considers objects tainted as a whole instead of tainting their fields. This approach is similar to our treatment for opaque objects.

In [25], authors modify the Python interpreter to provide a dynamic taint analysis for strings. More specifically, the representation of the class `str` is extended to include a boolean flag to indicate if a string is tainted. Similar to [15], our library supports taint analysis for several built-in types (e.g. strings and integers). The work by Seo and Lam [38], called InvisiType, aims to enforce safety checks (including taint analysis) without modifying the analyzed code. Their approach is designed for a stronger attacker model, i.e., an attacker that can have control over the source code. Therefore, InvisiType relies on several modifications in the Python interpreter in order to perform the security checks at the right places without the source code being able to jeopardize them. We consider a weaker attacker that only has control on input data and therefore no runtime modifications are required by our library.

Surprisingly, there is not much work considering taint analysis and persistent storage. In the information-flow community, Li and Zdancewic [26] enforce information-flow control in PHP where programs can use a relational database. The main idea of their work is to statically indicate the types of the input fields and the results of a fixed number of database queries. From a technical point of view, when type checking queries, their type-system behaves largely the same as when typing function calls. Rather than considering an static set of queries, our library is able to propagate taint information when entities are fetched from the datastore regardless the executed query. Another work dealing with persistent stores and information flow is the language Fabric [27]. Proposed by Liu et al., Fabric is essentially an extension to Jif [30] supporting distributed programming and transactions. Fabric allows the safe storage of objects, with exactly one security label, into a persistent storage consisting of a collection of objects. While Jif and Fabric are special purposes languages and our analysis works via a library, the manner that Fabric stores security labels in objects is similar to how taint information gets propagate into the GAE datastore.

The authors in [43] observe that sanitization should be context-sensitive, e.g., the sanitization requirements for a URI attribute are different from those of an HTML tag. In a similar spirit ScriptGard [37] automatically inserts, being context-sensitive, sanitizers in web applications to automatically repair vulnerabilities. In principle, it could be possible to implement some of those context-sensitive policies by enriching the information inside the attribute `taints` as well as the checks performed by the middleware when information is sent to the web server.

Considering a different setting, TaintDroid [18], a taint analysis for Android smartphones, tackles similar problems that the ones presented in this paper. In order to propagate taint information inside programs, TaintDroid requires the modification of the Android's VM interpreter. TaintDroid is also able to propagate taints tags (labels) into the file system extended attributes. To achieve that, a modification of the host file system (YAFFS2) is required. Our library, on the other hand, does not require the modification of the Python interpreter or the underlying datastore. The VM interpreter often calls native code which is unmonitored by TaintDroid. In a similar approach as for opaque objects, TaintDroid makes an approximation for the propagation of taint labels when calling native code, i.e., the label of the return value is the union of the taint labels of the call arguments. While the authors of TaintDroid manually patched native code to implement this approximation, our library provides a general method for approximating taints in opaque objects.

8. Final remarks and future work

We have developed a taint mode for the cloud computing platform Google App Engine for Python. Different from other taint analysis, our library propagates taint information into the datastore as well as opaque objects. We propose a security lattice as the general interface to specify interesting sanitization policies. Although this idea is not novel, we note that the least upper bound operation ($\sqcup$) is inadequate to describe the integrity level of aggregated data when using non-compositional sanitizers. The library defines default sources and sensitive sinks for the Google App Engine framework. In particular, it provides a middleware to intermediate between the web server and the application so that data obtained from the server gets automatically tainted as well as checks if the data being sent back is sanitized. Providing a WSGI-compliant middleware, the library can run with any web framework that follows the WSGI specification. We take a concrete example implementing a guestbook in the cloud and show how to adapt it to run our taint analysis by small modifications of the source code. We show that the library raises an alarm if developers do not sanitize data as indicated by the security policy.

There are several directions for future work. Focusing on avoiding XSS, the library declares the web server as a sensitive sink. However, we believe that there are other sensitive sinks in GAE. For instance, GAE applications can execute computational intensive numerical functions (e.g. through the `numpy` library), send emails [3] and even send HTTP requests [4] to other web sites. Evidently, user inputs or tainted data should not arbitrarily affect such operations. It would be interesting to develop a complete list of sensitive sinks for GAE. Other future work is to consider larger case studies for our library in order to determine the scalability of the approach. The code in the library is currently designed to be compact, easy to understand, and to be used during the development stage. It would be interesting to evaluate the performance of the library and introduce the required optimizations.

[3] https://code.google.com/appengine/docs/python/mail/

[4] https://code.google.com/appengine/docs/python/urlfetch/

Acknowledgments Thanks to David Sands, Andrei Sabelfeld, Bart van Delft, and Nick Johnson for valuable comments. This work was partially funded by the Google Research Award *Securing Google App Engine* and the Swedish research agency VR.

References

[1] OWASP Top 10 2010. `http://www.owasp.org/index.php/Top_10_2010`.

[2] The Perl programming language. `http://www.perl.org/`.

[3] The Ruby programming language. `http://www.ruby-lang.org`.

[4] CherryPy. `http://www.cherrypy.org/`.

[5] Django project. `http://www.djangoproject.com/`.

[6] Getting Started: Python - Google App Engine. `https://code.google.com/appengine/docs/python/gettingstarted/`.

[7] Samples for Google App Engine. `https://code.google.com/p/google-app-engine-samples`.

[8] Guetbook example for Google App Engine. `https://google-app-engine-samples.googlecode.com/files/guestbook_10312008.zip`.

[9] PEP 3333: Python Web Server Gateway Interface v1.0.1. `http://http://www.python.org/dev/peps/pep-3333/`.

[10] Pylons Project. `http://pylonshq.com/`.

[11] M. Andrews. Guest Editor's Introduction: The State of Web Security. *IEEE Security and Privacy*, 4(4):14–15, 2006.

[12] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2008.

[13] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, Apr. 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[14] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7.

[15] J. J. Conti and A. Russo. A taint mode for Python via a library. In *OWASP AppSec Research 2010. Invited paper to NORDSEC 2010*, LNCS, 2010.

[16] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[17] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[18] 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10. USENIX Association, 2010.

[19] Federal Aviation Administration (US). Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems, June 2009.

[20] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. In *Black Hat USA Briefings*, Aug. 2007.

[21] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[22] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. of the International Conference on World Wide Web*, May 2004.

[23] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

[24] J. Kong, C. C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06. ACM, 2006.

[25] D. Kozlov and A. Petukhov. Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. In *Proc. of Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, June 2007.

[26] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *Proc. of the 18th workshop on Computer Security Foundations*. IEEE Computer Society, 2005.

[27] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, , and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. ACM Symp. on Operating System Principles*, October 2009.

[28] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*. USENIX Association, 2005.

[29] M. Monga, R. Paleari, and E. Passerini. A hybrid analysis framework for detecting web application vulnerabilities. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IWSESS '09. IEEE Computer Society, 2009.

[30] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1999.

[31] National Institute of Standards and Technology. Definition of cloud computing. `csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`, 2011.

[32] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the Network and Distributed System Security Symposium*, 2005.

[33] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[34] T. Pietraszek, C. V. Berghe, C. V, and E. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, 2005.

[35] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21 (1):5–19, Jan. 2003.

[36] SANS (SysAdmin, Audit, Network, Security) Institute. The top cyber security risks. `http://www.sans.org/top-cyber-security-risks`, Sept. 2009.

[37] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11. ACM, 2011.

[38] J. Seo and M. S. Lam. InvisiType: Object-Oriented Security Policies. In *17th Annual Network and Distributed System Security Symposium*. Internet Society (ISOC), Feb. 2010.

[39] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09. ACM, 2009.

[40] A. van der Stock, J. Williams, and D. Wichers. OWASP Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`, 2007.

[41] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the Network and Distributed System Security Symposium*, Feb. 2007.

[42] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.

[43] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of XSS sanitization in web application frameworks. In *Proc. of the European Conference on Research in Computer Security*. Springer-Verlag, 2011.

[44] W. Xu, E. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical report, Stony Brook University, 2005.

[45] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*. USENIX Association, 2006.