Scalable Taint Specification Inference with Big Code

Victor Chibotaru
DeepCode AG, Switzerland
chibo@deepcode.ai

Benjamin Bichsel
ETH Zurich, Switzerland
benjamin.bichsel@inf.ethz.ch

Veselin Raychev
DeepCode AG, Switzerland
veselin@deepcode.ai

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

Abstract

We present a new scalable, semi-supervised method for inferring taint analysis specifications by learning from a large dataset of programs. Taint specifications capture the role of library APIs (source, sink, sanitizer) and are a critical ingredient of any taint analyzer that aims to detect security violations based on information flow.

The core idea of our method is to formulate the taint specification learning problem as a linear optimization task over a large set of information flow constraints. The resulting constraint system can then be efficiently solved with state-of-the-art solvers. Thanks to its scalability, our method can infer many new and interesting taint specifications by simultaneously learning from a large dataset of programs (e.g., as found on GitHub), while requiring few manual annotations.

We implemented our method in an end-to-end system, called Seldon, targeting Python, a language where static specification inference is particularly hard due to lack of typing information. We show that Seldon is practically effective: it learned almost 7,000 API roles from over 210,000 candidate APIs with very little supervision (less than 300 annotations) and with high estimated precision (67%). Further, using the learned specifications, our taint analyzer flagged more than 20,000 violations in open source projects, 97% of which were undetectable without the inferred specifications.

CCS Concepts • **Security and privacy** → **Information flow control**; • **Theory of computation** → *Program specifications*; • **Computing methodologies** → Semi-supervised learning settings.

Keywords Specification Inference, Taint Analysis, Big Code

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314648

ACM Reference Format:
Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable Taint Specification Inference with Big Code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3314221.3314648

1 Introduction

A wide class of security vulnerabilities occurs when user-controlled data enters a program via a library API (a *source*) and flows into a security-critical API (a *sink*). For example, a source may read a field in a web form or a cookie and a sink may write to arbitrary files on the server, execute arbitrary database commands (allowing sql injections) or render HTML pages (allowing cross-site scripting).

Failure to sanitize information flow from sources to sinks potentially affects the privacy and integrity of entire applications. As a result, there has been substantial interest in automated reasoning tools which detect such unsanitized flows, using both static and dynamic methods [18, 23, 27, 28]. While useful, all of these tools inherently rely on an existing specification that precisely describes the set of sources, sanitizers, and sinks the analyzer should consider. We refer to such a specification as a *taint specification*.

Key Challenge: Obtaining a Taint Specification. Obtaining such a specification for modern languages and libraries can be very challenging: to determine the role of a candidate API (source, sink, sanitizer, or none), one would have to inspect how information flows in and out of that API and how that information is used. Manually doing so over thousands of libraries with APIs often appearing in different contexts, is prohibitive.

To reduce this manual annotation burden, several works have proposed methods for automatically learning taint specifications (e.g., [4, 17, 24]). While promising, these works have several key drawbacks: either they work in a fully supervised manner requiring the user to manually annotate all sources and sinks in the training dataset [24], or work in a semi-supervised manner but do not scale beyond smaller programs [17], or require test cases and dynamic program instrumentation [4]. Further, all three of these methods target strongly typed languages (Java [4, 24] and C# [17]).

This Work: Scalable Semi-Supervised Learning. In this work we propose a new, scalable method for learning likely taint specifications in a semi-supervised manner. Given an input dataset of programs $\mathcal{D}$ where only a very small subset of the APIs $\mathcal{A}_M$ used by programs in $\mathcal{D}$ is manually annotated, our method infers specifications for the remaining, much larger un-annotated set of APIs $\mathcal{A}_U$ used by programs in $\mathcal{D}$. Unlike [24], to infer the taint specification for the APIs in $\mathcal{A}_U$, our approach leverages interactions among the APIs in $\mathcal{A}_U$ and $\mathcal{A}_M$ observed in the programs from $\mathcal{D}$.

The core technical idea of our approach is to formulate the taint specification inference problem as a linear optimization task, allowing us to leverage state-of-the-art solvers for handling the resulting constraint system. Importantly, unlike [17], our method scales to learning over large datasets of available programs $\mathcal{D}$ (*e.g.,* as found on GitHub), critical for increasing the number and quality of discovered specifications. To demonstrate its effectiveness, we implemented and evaluated our approach for Python, a dynamic language where discovering taint specifications is particularly challenging (*e.g.,* we can no longer rely on type information).

The overall flow of our end-to-end pipeline is shown in Fig. 1. The input to the training phase is a dataset of programs $\mathcal{D}$ where only a very small number of APIs $\mathcal{A}_M$ are annotated as sources, sinks, and sanitizers (in our evaluation, 106). We then apply our semi-supervised learning procedure based on the linear formulation, inferring 6 896 roles for the remaining APIs $\mathcal{A}_U$. These learned specifications $\mathcal{A}_U$ can then be examined by an expert or together with $\mathcal{A}_M$ be provided as input to a taint analyzer which will then search for vulnerabilities (a path between a source and a sink not protected by a sanitizer) based on the specifications.

Main Contributions. Our main contributions are:

- A formulation of the taint specification inference problem as linear optimization (§4).
- An end-to-end implementation of the inference method for Python in a tool called Seldon (§5).
- A thorough experimental evaluation indicating Seldon scales linearly with dataset size and can learn useful and new taint specifications with high precision: 67% on average (§7).
- A taint analyzer that uses the learned specifications to identify thousands of potential security violations in open-source projects. We examined some of these violations manually and disclosed them to maintainers of the code (§7). The tool is freely available[1] and is currently being used by Python developers in a push-button manner (about 1500 usages in January 2019).

[1] https://www.deepcode.ai/

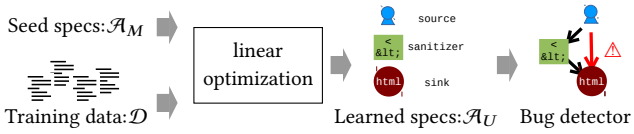


Figure 1. High-level overview of Seldon.

2 Overview

We now provide an informal explanation of our method on the example in Fig. 2. Full formal details are provided later.

Code Snippet. Fig. 2a shows a code snippet taken from a real code repository that we slightly adapted for brevity. It uses Flask, a Python microframework for web development and werkzeug, a web application library. The snippet (i) obtains the name of the file attached to a user POST request in Line 10, (ii) sanitizes the name in Line 11, and (iii) saves the file in Line 14. Proper sanitization in Line 11 is critical, as omitting it would enable attackers to use relative paths to store arbitrary files in arbitrary locations of the filesystem.

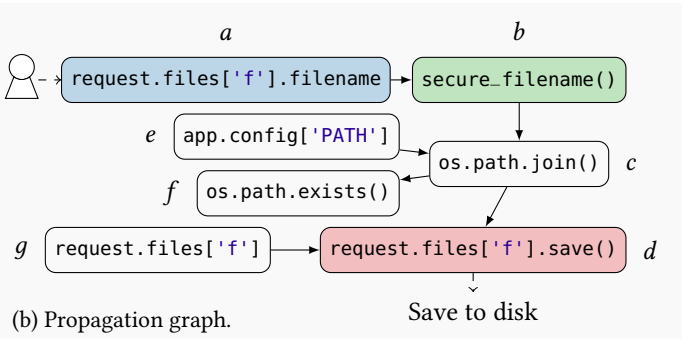
Taint Analysis. Fig. 2b shows the corresponding propagation graph which captures the information flow between different events in Fig. 2a. In addition to the graph, running taint analysis on the code snippet in Fig. 2a requires a *specification* of security-related events describing which events (i) represent user input (sources), (ii) sanitize user input (sanitizers), or (iii) must be protected by sanitizers to prevent vulnerabilities (sinks). In our example, `request.files['f'].filename` is a source (colored in blue), `secure_filename` is a sanitizer (colored in green), and `save` is a sink (colored in red). Given such a specification, taint analysis checks if there is information flow from a source to a sink that does not traverse a sanitizer, which amounts to a vulnerability.

Encoding Assumptions. The quality of a taint analyzer critically depends on the quality of its specification: an incomplete specification prevents detecting all vulnerabilities that involve missing parts of the specification. Seldon takes as input the propagation graphs of many applications, where the role of most events is unknown. For each event in the graph, it generates variables whose scores indicate the likelihood of this event having a particular role. For example, referring to event `request.files['f'].filename` as a, it introduces three variables, a^{src}, a^{san}, and a^{snk}. Seldon then searches for a joint assignment of scores to each variable. Because we restrict the scores of each variable to be between 0 and 1, we can interpret the resulting values as probabilities. For example, $a^{\text{src}} = 0.98$ means that event a most likely is a source.

As a next step, Seldon generates a constraint system that encodes reasonable assumptions on valid scores, shown in Fig. 2c. For example, constraint (1) encodes the belief that, if a is a source (*i.e.,* a^{src} is high), and b is a sanitizer (*i.e.,* b^{san} is high), then at least one of the events receiving information

```
1 from yak.web import app
2 from flask import request
3 from werkzeug import secure_filename
4 import os
5
6 blog_dir = app.config['PATH']
7
8 @app.route('/media/', methods=['POST'])
9 def media():
10   filename = request.files['f'].filename
11   filename = secure_filename(filename)
12   path = os.path.join(blog_dir, filename)
13   if not os.path.exists(path):
14     request.files['f'].save(path)
```

(a) Code snippet from real code.



(b) Propagation graph.

$$a^{\text{src}} + b^{\text{san}} \leq c^{\text{snk}} + f^{\text{snk}} + d^{\text{snk}} + C \quad (1)$$

$$a^{\text{src}} + c^{\text{san}} \leq f^{\text{snk}} + d^{\text{snk}} + C \quad (2)$$

$$b^{\text{san}} + d^{\text{snk}} \leq a^{\text{src}} + C \quad (3)$$

$$\cdots$$

(c) Constraints.

Figure 2. Example illustrating the workflow of Seldon.

flow from b must be a sink (*i.e.,* the sum of their sink scores must be high). In constraint (1), C is a constant that intuitively controls the strength of the implication (we use $C = 0.75$).

Likewise, constraint (2) enforces that if a is a source and c is a sanitizer, then one of f or d must be a sink. We note that, because c is not actually a sanitizer, constraint (2) should not be enforced. Because in this case, c^{san} is likely to be small, the constraint will indeed only have a minor effect on the solution to the constraint system.

Handling Ambiguous Targets of Events. Of course, if two events in different program locations target the same program element (*e.g.,* two function calls to the same function), we try to represent them with the same variables in the constraint system. However, in Python, this is non-trivial as it is hard to determine which function is targeted by a given function call. We elaborate more on this issue in Section §3.2, but provide a simple example here.

Concretely, a programmer might replace the two occurrences of `request.files['f']` in Fig. 2a by a function argument `f` instead, after replacing Line 9 by `def media(f):`.

Then, the target of event d is unclear (analogously for a). Seldon therefore collects two possible representations of d: r_1 = `media(param f).save()` (indicating function `save` of parameter `f` of `media`) and r_2 = `f.save()` (indicating function `save()` of `f`). The resulting constraint system is analogous to Fig. 2c, except for replacing variables by averages of all possible representations of their respective event. For example, we would replace d^{src} by $\frac{1}{2}\left(r_1^{\text{src}} + r_2^{\text{src}}\right)$.

If another source candidate event x in a different program location has two possible representation r_1 and r_3, the constraint system will use $\frac{1}{2}\left(r_1^{\text{src}} + r_3^{\text{src}}\right)$ to capture it, sharing variable r_1^{src} with d. As a consequence, the scores of x and d are correlated (increasing r_1^{src} increases the score of both x and d being a source).

Solving the Constraint System Resulting from Big Code. In this way, Seldon determines the propagation graphs for tens of thousands of Python source files, yielding roughly half a million constraints and 210,000 candidate events (for being sources, sanitizers, and sinks).

As programs in our training dataset need not always follow the intuitive assumptions on information flow, we may not be able to solve the constraint system from Fig. 2c exactly. Instead, we relax it, minimizing the violation of all constraints, while L_1-regularizing by the sum of scores (*i.e.,* favoring solutions with fewer inferred specifications).

3 Propagation Graphs

In this section we describe the concept of a propagation graph and discuss how to use it for analysis and learning.

3.1 Events and Information Flow

The propagation graph $G = (V, E)$ of a program P consists of a set of events V and a set of edges E capturing information flow, typically determined by static analysis. The set of events V captures all relevant actions in P that propagate information. Examples of events include function calls and global variable reads while examples of flows include the fact the return value of a function is used as an argument to another function. In practice, V is usually restricted to events relevant to a particular setting, *e.g.,* if we are interested in events relating to security, we can ignore actions that add

```
1 from base_driver import ThreadDriver
2
3 class ESCPOSDriver(ThreadDriver):
4   def status(self, eprint):
5     self.receipt('<div>'+msg+'</div>')
```

Figure 3. Example illustrating representation of function call events.

two integers. The graph G is built by applying a static information propagation analysis on P (*e.g.,* [17]) and thus edges represent an over-approximation of the actual information flow. To maintain consistency with standard notation for graphs, the set E captures information flow and not events (which are captured in V).

3.2 Representation of Events

An event $v \in V$ is captured by an expression in the program, for example a field read or a function call expression. To represent the possible targets an event v can resolve to, we define the function $\textsc{Rep}(v)$. For example, if v is a function call, $\textsc{Rep}(v)$ will be an expression describing functions that v can possibly resolve to.

Depending on the program and the type system of the language, the expression $\textsc{Rep}(v)$ will be of a different granularity. For instance, for a statically typed language, $\textsc{Rep}(v)$ may denote the exact function signature to be called by v, while for languages which lack static typing such as Python[2] it may be difficult to statically determine the possible targets of v, and hence $\textsc{Rep}(v)$ may denote an expression that captures a set of functions.

Example. In Fig. 3, the function call event at Line 5 will have a representation $\textsc{Rep}(v)$ given by:

```
ESCPOSDriver::status(param self).receipt()
```

However, we could also choose to use one of the alternative representations in the following list (ordered from most specific and precise to most general and least precise):

```
base_driver.ThreadDriver::status(param self).receipt()
status(param self).receipt()
self.receipt()
```

We have determined these options by (i) falling back to the base class of `ESCPOSDriver`, (ii) ignoring the class containing the function call, and (iii) ignoring the function containing the function call. Later, in Section §4.3, we will show how to (optionally) fall back to these alternative representations in order to improve the results of learning.

When building the propagation graph, even if two events v and v' have the same representation (*i.e.,* $\textsc{Rep}(v) = \textsc{Rep}(v')$), they will be encoded as separate events in V (*i.e.,* $v \neq v'$). Alternatively, if the representation is the same, we could collapse all events into one event (discussed in Section §6.4).

[2]Recently introduced gradual typing for Python allows type annotations for some variables, but is not yet widely used, so we do not rely on it.

3.3 Candidate Events and Roles

To enable taint analysis of G, we must assign a role (*i.e.,* source, sanitizer or sink) to each event in V. As in [17], we assume the role of an event only depends on its representation and hence it suffices to know the role of that representation. Consequently, when learning what the role of an event is, we are in fact learning what the role of its representation is. This is a slight simplification, *e.g.,* a function may act as a source or a sink depending on its arguments, however, we leave this differentiation for future work.

We only insert an event v into V if v is a candidate for a role and we also exclude specific roles for some events. For example, it is very rare that a field read acts as a sink and impossible for it to act as a sanitizer. Hence, field read events are only considered as candidates for sources. We elaborate more on this point in Section §5.

3.4 Using the Propagation Graph

Once the propagation graph is built (we discuss how to do so for Python in Section §5), we use this graph for two clients: specification learning, discussed in Section §4, as well as taint analysis for finding security issues (flows from sources to sinks which do not pass through a sanitizer). We note that for these two clients to be practically effective and produce useful results, it suffices for G to precisely capture the true information flow between events in *most* cases (that is, G need not be an over-approximation).

4 Learning Likely Taint Specifications

We now explain our method for learning roles of events in a given propagation graph G. The key insight is to formulate the specification inference problem as a linear optimization task that can be solved in a scalable manner with efficient solvers. The resulting scalability is important because it enables semi-supervised learning with very small amounts of supervision over a large dataset of programs. In turn, this leads to learning better specifications than if one performs inference over a single program only (as intuitively, the inference algorithm will then have access to only limited amounts of available information to learn from). We illustrate the benefits of our method experimentally in Section §7.

Learning over a Global Propagation Graph. In order to learn over a dataset of programs, we first extract the propagation graph $G_i = (V_i, E_i)$ from each program P_i in the dataset independently by using the method described in §3. The union of these individual graphs forms a global propagation graph $G = (\bigcup V_i, \bigcup E_i)$ over which the learning will be performed. Since the event sets V_i are pairwise disjoint (*i.e.,* $V_i \cap V_j = \emptyset$ for all $i \neq j$), G does not contain any edges between events from different programs. However, even when

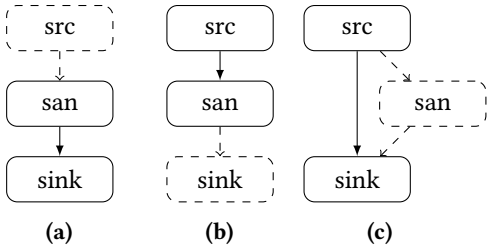


Figure 4. Visual illustration of our information flow constraints. Each graph states that if nodes (with solid outline) have the indicated roles, then there exists a node (with dashed outline) with the indicated role and information flow.

two events v and v' belong to different programs, they may share the same representation ($\textsc{Rep}(v) = \textsc{Rep}(v')$). Our learning method then leverages information across projects by mapping events with the same representation to the same variable (see §4.1).

We note that we explicitly allow events to have multiple roles (*e.g.,* source and sink), or no role at all, indicating the event is irrelevant for security (*e.g.,* the function `capitalize`).

Intuitive Information Flow Constraints. To perform learning, we first identify several constraints we believe should hold between events and which a solution of the learning task should try to enforce. An intuitive visualization of our constraints is provided in Fig. 4. Here, each column stands for a separate constraint and captures an implication. We note that our constraints are not the same as those of [17], however, they follow a similar high level idea of trying to capture information flow assumptions.

Fig. 4a indicates that if the program uses a sanitizer with information flow into a sink, this is most likely to sanitize a source. Therefore, it requires that whenever an event v has information flow into another event v', if we classify v as a sanitizer and v' as a sink, we must classify at least one of the events v_i with information flow into v as a source.

Analogously, Fig. 4b indicates that if the program uses a sanitizer which receives information from a source, the purpose of that sanitizer is to protect information flow into a sink. Finally, Fig. 4c indicates that a program typically sanitizes the information flowing into sinks, *i.e.,* if there is information flow from a source to a sink, there must be a sanitizer between these two events. As illustrated in Fig. 4c, we only require at least one path to be sanitized, not necessarily every path. This avoids (i) quantifying over all paths and (ii) being overly strict in the case of spurious information flow due to over-approximation.

Algorithmic Collection of Constraints. For a given propagation graph, we can easily identify all occurrences of the constraints in Fig. 4 by a variant of breadth-first search (BFS). For example, for Fig. 4a, we first identify all sanitizer candidates that flow into a sink candidate, by forward BFS starting from each sanitizer. For each such sanitizer v, we identify all source candidates v_i flowing into v. Constraint Fig. 4a then states that if v is a sanitizer and v' is a sink, then one of v_i must be a source.

4.1 Variables

For the representation $n = \textsc{Rep}(v)$ of some event $v \in V$, we introduce variables n^{src}, n^{san} and n^{snk} that encode likelihoods of n being a source, sanitizer, or a sink (we discuss multiple possible representations soon). To avoid unnecessary variables (and unnecessary constraints), we only introduce variables which match potential roles of n. For example, if n is a field read, we do not introduce the variable n^{san}. When convenient, we refer to variables treated as sources as $(n)^{\text{src}}$ and analogously for sanitizers and sinks.

Variable Constraints. To ensure that we can interpret each variable n^{role} as a probability, we add constraints

$$0 \leq n^{\text{role}} \leq 1 \tag{4}$$

for each introduced variable n and each role in {src, san, snk}.

Constraints for Known Variables. In our evaluation, we label a small number of events by hand so to bootstrap learning (hence, our method is semi-supervised), accomplished by constraining some variables to have appropriate roles. For example, when we label an event as a sanitizer, but not as a source or a sink, we add the constraints $n^{\text{src}} = 0$, $n^{\text{san}} = 1$, and $n^{\text{snk}} = 0$, where n is the representation of that event.

4.2 Linear Formulation of Information Flow

We now discuss how to encode our intuitive assumptions on information flow from Fig. 4 as linear constraints. As we will see, formulating the meaning of the intuitive assumptions as linear constraints is crucial in order to ensure the resulting constraint system is efficiently solvable (and thus, suitable for learning from a large dataset of programs).

Linear Formulation of Fig. 4a. To express the assumption in Fig. 4a as a linear constraint, we assert that for every event v_{sanit} that may act as a sanitizer and flows into a potential sink event v_{sink}, the sum of all source candidate scores flowing into it must be at least the sum of scores of v_{sanit} and v_{sink}.

Example. Fig. 5 shows a small propagation graph for which we demonstrate the constraint derived from the assumption in Fig. 4a. Let v_{sanit} be an event with representation $n_{\text{sanit}} = \textsc{Rep}(v_{\text{sanit}})$, which may act as a sanitizer. Analogously, let v_1, v_2, v_3 be events with representations n_1, n_2, n_3, which may act as sources, and v_{sink} be an event with representation n_{sink} that may act as a sink. Further, we assume that the unlabeled event (top left) may not act as a source (hence we ignore it). Then, to encode the assumption from Fig. 4a, we add the following constraint

$$(n_{\text{sanit}})^{\text{san}} + (n_{\text{sink}})^{\text{snk}} \leq (n_1)^{\text{src}} + (n_2)^{\text{src}} + (n_3)^{\text{src}} + C \tag{5}$$

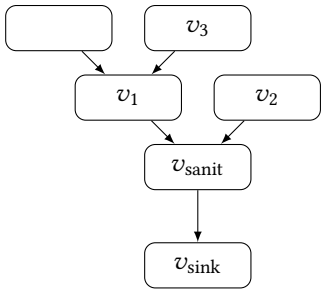


Figure 5. A propagation graph used to illustrate the linear constraints we derive to formally capture Fig. 4a.

where C is a fixed constant. If we set $C = 1$ and restrict the values of variables $(n_1)^{\text{src}}$, $(n_2)^{\text{src}}$, $(n_3)^{\text{src}}$, $(n_{\text{sanit}})^{\text{san}}$, and $(n_{\text{sink}})^{\text{snk}}$ to $\{0, 1\}$, then constraint (5) precisely captures the assumption from Fig. 4a: first, if $(n_{\text{sanit}})^{\text{san}} = (n_{\text{sink}})^{\text{snk}} = 1$, then at least one of $(n_1)^{\text{src}}$, $(n_2)^{\text{src}}$ and $(n_3)^{\text{src}}$ must be 1. Second, constraint (5) is trivially satisfied if at least one of $(n_{\text{sanit}})^{\text{san}}$ or $(n_{\text{sink}})^{\text{snk}}$ is 0.

Restricting values of variables to $\{0, 1\}$ would result in a (generally NP-complete) integer linear program and induce an unsatisfiable constraint system. Instead, we relax the values a variable can take to be in the interval $[0, 1]$, as indicated in constraint (4). Then, solving the resulting large constraint system spanning many variables, becomes tractable. We can interpret constraint (5) for the relaxed constraint system by observing that if $(n_{\text{sanit}})^{\text{san}}$ and $(n_{\text{sink}})^{\text{snk}}$ are large, then at least one of $(n_1)^{\text{src}}$, $(n_2)^{\text{src}}$, $(n_3)^{\text{src}}$ must be large as well.

Empirically, we observed that for $C = 1$, most scores are quite close to 0, precluding a clear separation of sources, sinks, and sanitizers. Therefore, we decreased C to 0.75, which increases the effect of the constraints when variables on the left-hand side have scores below 1. For example, if $(n_{\text{sanit}})^{\text{san}} = (n_{\text{sink}})^{\text{snk}} = 0.7$, then $C = 0.75$ ensures that $\sum_{i=1}^{k} n_i^{\text{src}} \geq 0.65$, whereas $C = 1$ only ensures $\sum_{i=1}^{k} n_i^{\text{src}} \geq 0.4$. To avoid overfitting, we only tested $C = 0.75$, concluding it performs significantly better than $C = 1$.

Linear Formulation of Fig. 4b. To express the assumption from Fig. 4b, consider a potential source event v_{source} with representation n_{source}, a potential sanitizer event v_{sanit} with representation n_{sanit}, and potential sink events $v_1, \ldots, v_k$ with representations $n_1, \ldots, n_k$, respectively. In addition, assume there is information flow from v_{src} to v_{sanit} and from v_{sanit} to every v_i. Then, we encode Fig. 4b via

$$(n_{\text{source}})^{\text{src}} + (n_{\text{sanit}})^{\text{san}} \leq \sum_{i=1}^{k} n_i^{\text{snk}} + C \tag{6}$$

The interpretation of constraint (6) is analogous to constraint (5).

Linear Formulation of Fig. 4c. We derive the remaining constraint analogously, leading to

$$(n_{\text{source}})^{\text{src}} + (n_{\text{sink}})^{\text{snk}} \leq \sum_{i=1}^{k} (n_i)^{\text{san}} + C \tag{7}$$

4.3 Selecting Event Representations with Backoff

In programming languages without static type information (in our case Python), it is typically difficult to determine which possible targets an event v can resolve to, as discussed earlier in §3.2. At that point, we had derived the representation $\textsc{Rep}(v)$ of a particular function call event as

```
ESCPOSDriver::status(param self).receipt()
```

However, using this entire expression (which captures the exact location of `receipt`) may be too detailed and overly specific to only a particular event, leading to a representation $\textsc{Rep}(v)$ that occurs infrequently. To address this issue, we can fall back to less specific representations that occur more frequently. For example, there may be many more calls to `receipt()` from other subclasses of `base_driver.ThreadDriver` to `receipt()`, in which case it would make sense to fall back to the following representation

```
base_driver.ThreadDriver::status(param self).receipt().
```

However, falling back to a less specific representation induces the risk of conflating $\textsc{Rep}(v)$ with other, unrelated occurrences of `self.receipt()` from unrelated events at other locations. As a trade-off between both approaches, that is, working with a long but infrequent representation vs short but frequent representation, we use a backoff approach that takes into account all possible suffixes (until we reach a frequent suffix). This is a commonly used approach in machine learning, see *e.g.,* [3], which we adapt to our setting.

Selecting of Backoff Variables. Concretely, let $(n_1, \ldots, n_k)$ be all potential representations of an event v, ordered from most to least specific. If some representations $\{n_1, \ldots, n_l\}$ occur infrequently, we will ignore them as these do not provide much information (in our evaluation, the cutoff threshold is 5 occurrences). We denote the remaining set of all possible backoff options for v by $\textsc{Reps}(v) = \{n_{l+1}, \ldots, n_k\}$. If no potential representation of v occurs frequently enough (*i.e.,* if $\textsc{Reps}(v) = \emptyset$), then we also ignore v and all constraints involving v, as we cannot hope to learn much about v.

Overall, instead of creating only 3 variables n^{src}, n^{san}, and n^{snk}, we create $3 \cdot |\textsc{Reps}(v)|$ variables for every possible backoff option

$$\begin{array}{ll} (n_r)^{\text{src}} & \forall r \in \textsc{Reps}(v) \\ (n_r)^{\text{san}} & \forall r \in \textsc{Reps}(v) \\ (n_r)^{\text{snk}} & \forall r \in \textsc{Reps}(v) \end{array}$$

Variable Constraints. To interpret values of these variables as probabilities (cp. §4.1), we again add the constraint

$$0 \leq (n_r)^{\text{role}} \leq 1 \quad \forall r \in \textsc{Reps}(v), \text{role} \in \{\text{src}, \text{san}, \text{snk}\} \tag{8}$$

Information Flow Constraints. Finally, for every constraint we added in §4.2, we introduce a constraint that replaces each variable $(n)^{\text{role}}$ by an average of its backoff options

$$\frac{1}{|\textsc{Reps}(v)|} \sum_{r \in \textsc{Reps}(v)} (n_r)^{\text{role}}$$

4.4 Relaxing and Solving the Constraint System

We now discuss how to solve the resulting constraint system derived by applying backoff (§4.3) to variables and constraints introduced in §4.1 and §4.2, respectively.

The constraint system C consists of variable constraints C^{var} as in constraint (8), constraints for known variables C^{known} as in §4.1 and information flow constraints C^{flow} as in §4.2. For all constraints, we use the version which includes backoff (§4.3), but for C^{known}, we only annotate fully qualified names, *i.e.,* the longest possible backoff option for each event.

Relaxing the Constraints. Because programs in our training dataset might not always respect our intuitive constraints on information flow, the constraint system is generally not satisfiable. For example, the assumption from Fig. 4a might be violated by a sanitizer that does not have any sources flowing into it, which in turn would violate constraint (5) and hence also its version with backoff.

This forces us to relax the constraints and replace the entire constraint system with a more robust one, which takes into account possible errors. Intuitively, we treat C^{var} as hard constraints but relax C^{flow} to be soft constraints, and aim to minimize their violation.

More precisely, we transform each constraint in C^{flow} of the form $L_i \leq R_i$ into a relaxed constraint $L_i \leq R_i + \varepsilon_i$, where L_i (resp. R_i) is the left-hand (resp. right-hand) side of the original constraint, and $\varepsilon_i \geq 0$ quantifies the violation of the original constraint.

Rewriting this constraint yields $L_i - R_i \leq \varepsilon_i$. Hence, minimizing the violation of the original constraints (measured by the l^1-norm defined as $\sum_i |\epsilon_i|$) means solving the following optimization problem

$$\min \sum_{i=1}^{M} \max(L_i - R_i, 0) + \lambda \cdot \sum_{n_r} \left((n_r)^{\text{src}} + (n_r)^{\text{san}} + (n_r)^{\text{snk}}\right) \tag{9}$$

$$\text{subject to } 0 \leq (n_r)^{\text{role}} \leq 1 \quad \forall n_r \text{ and} \tag{10}$$

$$C^{\text{known}} \tag{11}$$

Here, n_r ranges over all representations and their backoff versions, and M denotes the total number of constraints in C^{flow}. In addition, λ is a regularization constant (in our evaluation, we use $\lambda = 0.1$) that penalizes solutions that classify many nodes as sources, sanitizers or sinks. To avoid overfitting, we tried only a few values $\lambda \in \{0.1, 1, 0.01\}$, of which $\lambda = 0.1$ performed best. Empirically, we observed that decreasing λ by a factor of 10 increases the number of inferred specifications by a factor of around 2.

Solving the Relaxed Constraint System. Solving the resulting optimization problem (9) efficiently can be done in many different ways. We opted for using Adam Optimizer [15] from TensorFlow [1], which optimizes (9) incrementally using projected gradient descent. Here, to enforce the hard constraints of type (10), we project the variables n^{role} to the interval $[0, 1]$ after each iteration.

5 Building the Propagation Graph for Python

To build the propagation graph $G = (V, E)$ of a Python program P, we first determine the set of relevant events V and their candidate roles, and then establish the information flow between these events.

5.1 Events

The events in our propagation graph can be either function calls, object reads, or formal arguments of function definitions. Examples of object reads include attribute loads (*e.g.,* `obj.field`), indexing into dictionaries, lists, and tuples (*e.g.,* `d[k]`), and reads of method parameters (*e.g.,* function `def f(a)` has potential source `a`). As mentioned earlier, the role of an object read event can only be a source, but not a sanitizer or a sink. Similarly, formal arguments of function definitions are considered only as sources. Finally, function calls can be either sources, sinks or sanitizers. We do not consider stores into variables because they rarely act as sinks and cannot act as a source or a sanitizer.

5.2 Capturing Information Flow

We next discuss how we capture information flow.

Function Calls. We assume a function call induces information flow from its arguments to its return value, an overapproximation of the true information flow, *e.g.,* the Python built-in function `id(o)` returns a unique value for object `o`, and hence does not propagate information about `o` itself.

Points-to Analysis. Points-to analysis (also known as alias analysis) determines which variables may point to the same object. In our system we used an Andersen-style points-to algorithm [26]. Here, for each pair of events a, b s.t. $a \in \mathit{PointsTo}(b)$ we add an edge $b \longrightarrow a$. Our points-to algorithm is fully flow-sensitive, field-sensitive and context-sensitive up to a bound of 8 method calls, the default setting of the library we used. For simplicity, we treat calls to functions with an unknown body as allocation sites and ignore loops (*i.e.,* we treat loops as having just a single iteration). As a consequence, our propagation graph does not contain cycles (in principle, our method supports cycles).

We note that our points-to analysis both over- and under-approximates the true points-to relation but is sufficiently

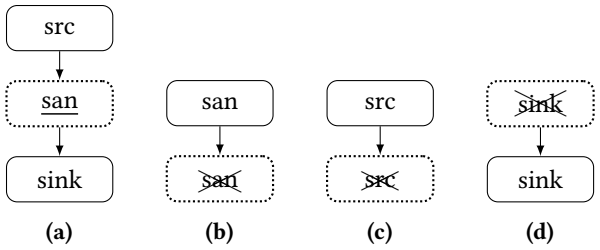


Figure 6. Visual illustration of Merlin's information flow constraints. Each graph states that if solid nodes have the indicated roles, the dotted node should have the indicated role (if underlined) or not have it (if crossed out).

precise for our application of learning role specifications of events and the subsequent taint analysis that uses these specifications to discover security bugs.

Data Structures. Python has several built-in collections (*i.e.,* data structures), including *lists*, *dictionaries*, *tuples*, and *sets*. We treat these the same as standard function calls, *e.g.,* when defining a new list (*e.g.,* `l=[a,b,c]`), we record information flow from any of its entries to the whole list.

Local Symbol Table. The function *locals()* returns a dictionary containing all of the local variables in the current scope. We model this behavior by adding information flow from all local variables to *locals()* calls.

Inlining Methods. Finally, to enable a more precise static analysis, we inline methods whose body can be statically determined (*i.e.,* methods defined in the same file as the caller and not subject to multiple dispatching). For simplicity, we treat imported methods as having an unknown body.

6 Adapting a Baseline Method

We now discuss Merlin's method to taint specification inference in C# [17]. Merlin's method differs from our method across several key technical dimensions: (i) the formulation of the constraints, (ii) how relevant events and candidate roles are identified, (iii) the formulation of the optimization problem, and (iv) the granularity of the propagation graph.

6.1 Merlin's Constraints

We summarize the information flow constraints of Merlin in Fig. 6 (their presentation in [17] is different, but equivalent).

Fig. 6a illustrates that for every flow across three events where the first is a source and the third is a sink, the second is likely to be a sanitizer. Fig. 6b shows that whenever there is a flow from a sanitizer to another event, the latter is unlikely to be a sanitizer. Fig. 6c and Fig. 6d capture an analogous constraint for sources and sinks.

In contrast to our constraints, Merlin's constraints cannot be easily encoded as linear constraints. In addition, Merlin's constraints are more restrictive: they constrain the role for specific nodes, while our constraints only enforce the existence of nodes with a particular role.

6.2 Identifying Events and Candidate Roles

Merlin relies on static typing information when identifying events and candidate roles. Since this is impossible in programming languages without strong static typing, we adapted Merlin to handle such languages.

Representation of Events. As Merlin targets C#, it does not address how to deal with multiple possible targets of a given event. Instead, it represents events by their target, identified using the static type system. For dynamically typed languages, this is hard, as discussed in §3.2. Thus, when adapting Merlin to handle Python programs, we use the most specific representation (*e.g.,* the one discussed earlier for the example in §3.2).

Identification of Candidates. To identify which nodes are candidates for a role, Merlin heavily relies on the type system of C#. For example, it considers only functions that take a string as input as candidates for sources. Applying this heuristic to languages without static type information is not possible which means we can exclude only very few candidates from certain roles. For example, we view all function calls as candidates for all roles. As a consequence, in our setting, we obtain significantly more candidates than Merlin does in C#, but typically the same amount of information about their interaction. This increases the difficulty of specification learning because we now have to classify more candidates using roughly the same amount of information.

6.3 Formulating the Optimization Task

We now discuss how Merlin uses its information flow constraints to state an optimization problem on factor graphs. As we demonstrate in §7.4, in contrast to our approach based on linear constraints, this formulation does not scale well and is not suitable for learning from a large dataset of programs.

Factor Graphs. Merlin expresses its information flow constraints as a factor graph, a well-known graphical model [16] suitable for performing structured prediction.

Given the propagation graph G of a program, Merlin scores joint assignments of roles to events. To this end, it introduces a score for every occurrence of a constraint from Fig. 6 in G. For example, based on the constraint in Fig. 4c, Merlin investigates every flow through three events which are candidates for being sources, sanitizers, and sinks, respectively. For each such flow, Merlin checks if the joint assignment respects the constraint: if the first and third events are classified as source and sink, respectively, the second event should be classified as a sanitizer. If so, it will provide a high score, and if not, a low score.

Figure 7. Illustrating vertex contraction: merging gray nodes.

Then, the total score of an assignment is the product of all scores

$$p(x_1, \ldots, x_N) = \prod_s f_s(x_s) \tag{12}$$

Here, each $x_i \in \{0, 1\}$ encodes if a given event assumes a given role. Moreover, each f_s is a factor evaluating the score for an occurrence of a constraint in G, x_s is the set of variables involved in this factor, and $f_s(x_s)$ is the score of factor f_s for assignment x_s.

Prior on Known Candidates. For candidates that are labeled by hand, Merlin selects a hard prior, stating that they must act as their respective role. We can interpret this as a particular factor for every hand-labeled candidate, defined by

$$f_s(x_s) = \begin{cases} 1 & \text{candidate has the correct role} \\ 0 & \text{otherwise} \end{cases}$$

Prior on Candidates. In addition to scores for joint assignments, Merlin introduces prior probabilities for every potential role of every candidate node. The probability a source (resp. sink) candidate acts as a source (resp. sink) is 50%. The probability a sanitizer candidate acts as a sanitizer is intuitively given by checking which fraction of paths that go through it start from a source and end in a sink. These priors can be seen as additional factors, meaning that Eq. (12) can accurately describe the joint probability of all factors.

Probabilistic Inference. Using prior probabilities and the scores for joint assignments, Merlin determines the marginal probability of each candidate having a particular role:

$$p_i(x_i) = \sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_N} p(x_1, \ldots, x_N) \tag{13}$$

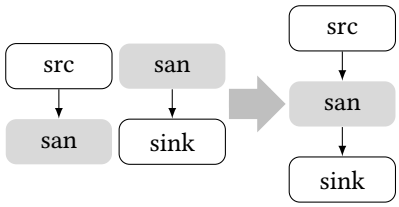
While there is a wide variety of techniques that solve marginal inference in factor graphs (*e.g.,* the sum-product algorithm [31]), these are significantly less scalable than approaches for solving linear constraints (which our approach uses). To solve Eq. (13), Merlin uses Infer.NET [19].

6.4 Propagation Graph Granularity

In contrast to our approach, Merlin conflates events targeting the same AST node (*i.e.,* events with the same representation). Because our propagation graph is more informative than Merlin's, we can use it to construct the conflated graph by applying vertex contraction to conflate all events with the same representation, illustrated in Fig. 7. We refer to the original graph as *uncollapsed*, and to Merlin's propagation graph as *collapsed*.

```
1 def f():
2   x = src()
3   y = san(x)
4 def g():
5   x = 1
6   y = san(x)
7   sink(y)
```

(a) Two functions propagating information.



(b) Transition from uncollapsed (left) to collapsed (right) propagation graph.

Figure 8. Collapsed graphs are unsuitable for taint analysis.

The collapsed graph is not suitable for taint analysis as it combines unrelated events into a single graph, illustrated in Fig. 8: here the collapsed graph contains information flow from the source to the sink even though this is not the case in the original program. We note that while the collapsed graph is not suitable for taint analysis, it can still be used for specification learning.

While Merlin's original presentation [17] assumes collapsed graphs, it is also directly applicable to uncollapsed graphs. Therefore, in our evaluation, we run Merlin using both the collapsed and uncollapsed propagation graphs.

7 Experimental Evaluation

We now present an extensive experimental evaluation of Seldon[3], and demonstrate its practical effectiveness in both learning useful taint specifications from a large dataset of Python programs and finding new security bugs based on these learned specifications.

7.1 Implementation and Analysis Flow

We implemented Seldon in an end-to-end production quality tool which (i) parses and analyzes a large number of Python repositories, (ii) extracts the propagation graphs from programs as outlined in Section §5, (iii) builds the resulting linear constraint system as outlined in Section §4, and (iv) solves the optimization problem subject to the constraint system.

Once the optimization problem is solved, we obtain confidence scores for every possible backoff option of every event being a source, sink, or sanitizer. To determine the role of a given event, we loop over all its possible backoff options, sorted from most to least specific. For the i^{th} option (0-based), we select a given role if $0.8^i \cdot (n_i)^{\text{role}} \geq t$, for a given threshold t (discussed later). It this does not happen for any role or backoff option, the event has no role. Overall, this amounts

[3]Seldon is freely available at https://www.deepcode.ai/

Table 1. Statistics on the applications in our evaluation.

# Candidates	210 864
Average # backoff options per event	1,73
# Constraints	504 982
# Source files	44 250

to backing off to less and less specific representations while exponentially decreasing the score.

We also built a static taint analyzer that consumes the specification learned by Seldon and checks for security vulnerabilities in web applications: it reports an error if it finds an unsanitized information flow from a source to a sink.

7.2 Dataset

In this section, we describe the programs and hand-labeled specifications used for our evaluation.

Applications. We obtained various Python applications from GitHub (https://github.com/), focusing on popular ones (identified by number of stars). While Seldon is generally applicable, we focused our attention on web applications, a domain where practical security concerns are particularly relevant. To this end, we only included applications which contain the string `django`, `werkzeug` or `flask` in their code, as these are the most popular frameworks used for Python web development. Tab. 1 provides statistics on these applications, including the number of candidates for sources, sanitizers, and sinks, the average number of backoff options per event, the number of constraints induced by the applications, and the number of Python files.

Seed Hand-labeled Specifications. We manually annotated some events, resulting in 106 events classified as exclusively sources (28 events), exclusively sanitizers (30 events), and exclusively sinks (48 events). To avoid issues with events with multiple possible representations (see Section §4.3), we only classified events whose representations are fully qualified, such as `werkzeug.utils.secure_filename()`. We avoided labeling application-specific events or other common coding patterns, ensuring that we can handle new programs without needing additional manual effort.

In general, manually annotating events can be expensive. However, we focused our attention on events with a clear role (*e.g.,* `request.GET.get()`), identified by investigating common security vulnerabilities (as taken from OWASP Top 10 [22]). This approach allowed us to create our seed specification within a few hours, labeling most events involved in well-known taint-related security vulnerabilities. Labeling more events is significantly harder as it requires manually identifying and investigating less common vulnerabilities.

We also blacklisted some built-in Python functions and commonly used library functions from taking on any role, by specifying 192 patterns, including, *e.g.,* `*__name__*`, `*.all()`, or `*tensorflow*`. This blacklist contains patterns for commonly used Python code patterns. The seed specification is given in App. B.

$$
\begin{aligned}
v_{\text{src}} \text{ source} &: \exists v_{\text{sink}}.\exists P.(v_{\text{src}}, v_{\text{sink}}, P) \text{ vulnerable} \\
v_{\text{san}} \text{ sanitizer} &: \exists v_{\text{src}}.\exists v_{\text{sink}}.\exists P.(v_{\text{src}}, v_{\text{sink}}, P) \text{ vulnerable} \\
&\quad \text{and } (v_{\text{src}}, v_{\text{san}}, v_{\text{sink}}, P) \text{ sanitized} \\
v_{\text{sink}} \text{ sink} &: \exists v_{\text{src}}.\exists P.(v_{\text{src}}, v_{\text{sink}}, P) \text{ vulnerable}
\end{aligned}
$$

Figure 9. Formal definitions for sources, sanitizers and sinks.

7.3 Manually Inspecting Learned Specifications

For both Seldon and our baseline Merlin, we investigated the precision of the learned specifications.

Ground Truth. To determine the ground truth, we defined sources, sinks, and sanitizers via a notion of vulnerable information flow, in a given program. Concretely, we say a triple (v, v', P) is vulnerable, if an attacker can inject malicious data into v, which then flows into v', which in turn triggers a vulnerability (*e.g.,* an SQL injection). We note that in this case, we ignore sanitizers on the path from v to v', because we are only interested in determining if the investigated information flow is potentially dangerous. Conversely, we define a quadruple $(v, v_{\text{san}}, v', P)$ as sanitized, if sanitizer v_{san} safely sanitizes the information flow from v to v' in P.

The formal definitions for the considered roles are shown in Fig. 9. Here, quantifiers expand to programs and events from our dataset, ensuring we only report sources, sinks, and sanitizers that may lead to vulnerabilities in real-world programs.

Finding Vulnerable Flows. We note that determining if a program and an event with the specified properties exist would generally require investigating all programs in our dataset. Because this is practically infeasible, we instead use taint analysis (Section §3.4) to determine candidate programs to investigate. For example, to evaluate if a given event n is a source, we run taint analysis on all programs to identify all sinks that are reachable from this source and contained in our specification. Then, we only investigate the top 5 sinks with the highest scores. If none of them exhibits a vulnerable flow, we (conservatively) report n as a false-positive.

Fairness of Comparison. To ensure fairness when comparing Merlin to Seldon, we evaluate both specifications in exactly the same way, using the whole dataset and the candidates inferred by our approach to identify candidates for vulnerable programs. We note that the alternative of using only the dataset and specifications used and obtained by Merlin would be unfair to Merlin, as it only scales to a single application, and thus predicts significantly less specification candidates.

Table 2. Statistics on specification learning with Merlin.

Repository	Number of Lines	Graph type	Candidates (src/san/sink)	Factors	Inference Time
Flask API	2 128	Collapsed	376/376/68	1034	2min
Flask API	2 128	Uncollapsed	378/378/66	470	3min
Flask-Admin	23 103	Collapsed	3337/3337/422	22740	> 10h
Flask-Admin	23 103	Uncollapsed	3369/3369/380	7204	> 10h

Table 3. Results for Merlin on the Flask API, only selecting roles with a confidence of 95%.

	Collapsed Graph		Uncollapsed Graph	
Role	Number	Precision	Number	Precision
Sources	18	33%	9	22%
Sanitizers	5	20%	1	100%
Sinks	3	0%	3	0%
Any	26	27%	13	23%

Table 4. Results for Merlin on the Flask API, only selecting top 5 predictions.

	Collapsed Graph		Uncollapsed Graph	
Role	Number	Precision	Number	Precision
Sources	5	40%	5	20%
Sanitizers	5	20%	5	40%
Sinks	5	0%	5	0%
Any	15	20%	15	20%

7.4 Baseline Comparison

To evaluate Merlin [17] we used the adaptation in Section §6 and the same propagation graph, seed specifications, and set of candidates as for the evaluation of our approach. We specified the factor graph derived from the propagation graph in Infer.NET [19]. As Merlin originally specifies a collapsed propagation graph, while our approach specifies an uncollapsed one, we evaluated Merlin on both cases.

Because the employed inference method is not specified in [17], we used the default method of Expectation Propagation (EP), including belief propagation as a special case. We also tried Gibbs sampling when EP timed out. We note that the third available inference method, Variational Message Passing, could not handle the encoded factor graph.

We ran all experiments for Merlin on a 512GB RAM, 32 core machine at 2.13GHz, running Ubuntu 16.04.3 LTS.

Scalability. Merlin was unable to handle the large amount of data we use for our learning approach. This is not entirely surprising, as it relies on probabilistic inference, a task that is #P-hard when exact results are desired [6], and NP-hard even for approximate results [7].

We ran Merlin on two applications, summarized in Tab. 2: Flask API [10], which is smaller and where inference succeeds in minutes, and the larger Flask-Admin [9], where inference timed out after 10h. We also tried Gibbs Sampling with only 20 iterations, which also timed out after 10h. In contrast, our approach handles Flask-Admin in < 20 seconds (we did not evaluate the precision here as we scale to the full dataset). As a consequence, running Merlin on the whole dataset used for the evaluation of our approach is infeasible.

When Merlin did not time out (*i.e.,* for Flask API), we investigated its precision using the method from Section §7.3. Tab. 3 shows the results when selecting using a threshold of 95%, *i.e.,* only picking roles for which Merlin is highly confident. The results indicate Merlin is often overly confident, but not very precise. This is the case regardless of whether the graph is collapsed or not. The precision results are analogous when examining its top 5 predictions (Tab. 4).

Because Merlin predicted only few specifications in both experiments, we also investigated the effect of increasing the number of predicted specifications. Concretely, we computed Merlin's estimated precision with a threshold of 50% for sinks and sanitizers (yielding less than 50 specifications) and for the top 50 sources, corresponding to a threshold of 50% (54%) for the collapsed (uncollapsed) graph. For the collapsed (uncollapsed) graph, the precision is 18% (12%) for sources, 2.7% (2.22%) for sinks, and 8.33% (9.09%) for sanitizers.

Based on these experiments, we concluded that Merlin cannot reach high precision in our setting.

7.5 Evaluation Results for Seldon

To evaluate Seldon, we investigated the following questions:

- **Q1**: How scalable is our approach in solving its constraint system?
- **Q2**: How precise is the learned specification, *i.e.,* how many identified roles are false positives?
- **Q3**: How many sources, sanitizers, and sinks do we detect?
- **Q4**: How useful are learned specifications for discovering vulnerabilities?
- **Q5**: Does using a large dataset improve the inferred specifications?
- **Q6**: How does the seed specification affect precision?
- **Q7**: What is the real-world value of the security vulnerabilities reported by Seldon?

Experiments were done on a 28 core Intel(R) Xeon(R) CPU E5-2690 v4, 512Gb RAM machine running Ubuntu 16.04.3.

Table 5. Count and estimated precision of candidates predicted by Seldon.

Role	# Predicted / # Candidates	Fraction of predicted candidates	Precision (Estimate)
Sources	4 384 / 210 864	2.08%	72.0%
Sanitizers	1 646 / 210 864	0.78%	58.0%
Sinks	866 / 210 864	0.41%	56.0%
Any	6 896 / 210 864	3.27%	66.6%

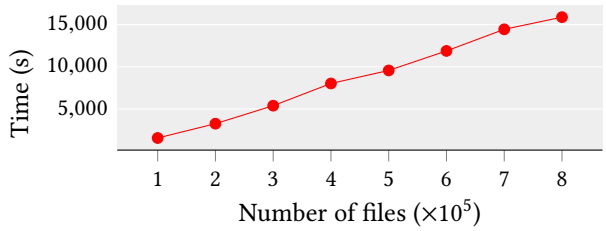


Figure 10. Seldon inference times (in seconds) as a function of the number of analyzed files.

Q1: Scalability. We tested the scalability of Seldon by running it on datasets of different sizes. The measured inference times are shown in Fig. 10, showing Seldon scales linearly in the number of analyzed files (the graph looks similar when we consider the size of the constraint system instead of the number files). For the largest dataset of 800,000 files (including non-web applications), Seldon finished in < 5 hours, indicating it is scalable enough to learn specifications from real-world repositories.

Q2: Precision. We have evaluated the precision of predicted sources, sinks, and sanitizers, *i.e.,* which fraction of candidates that are predicted to have a particular role actually have the predicted role. To perform this evaluation, we took a random sample of 50 sources, sanitizers, and sinks with scores above 0.1. To determine this threshold for sources (analogously for sinks and sanitizers), we sorted events by their score for being a source and picked the threshold that strikes a balance between number of predicted specifications (*i.e.,* recall) and precision. Concretely, we tolerate a slightly higher false positive rate for the opportunity to discover new unknown security vulnerabilities.

We manually determined the ground truth for these events, leading to a precision of 72.0% (sources), 56.0% (sinks) and 58.0% (sanitizers), respectively, and an overall precision of 66.6% (see also Tab. 5). We have repeated this experiment for more samples (manually checking 200 sources, sanitizers, and sinks, respectively), yielding an overall precision of 65.5% (this corresponds to a deviation of 1.1%). This indicates that our precision estimate is fairly stable.

In Fig. 11, we provide more details on the score and precision of the random sample we evaluated. First, note there are only a few samples with score around 1.0, meaning the optimization was completely confident in its assignment of a role for relatively few events. For most samples, the score is around 0.5, hence the model believes the event likely plays the role, but is uncertain. Fig. 11 also shows how confidence transforms into precision. For every precision data point, we show the average ratio of true positive labels according to the ground truth for the cumulative sample including all samples up to a position. This shows how we can adjust the threshold for selecting sources, sinks and sanitizers to achieve desired precision levels. Note, however, that non-zero confidence value for a role is already an indicator for the event's role, as most events (discussed in Q3) have score 0.

Our results indicate that Seldon successfully identifies many sources, sanitizers, and sinks with consistently high precision. App. A shows the evaluated sample of events.

Table 6. Results for bug-finding using the set of sources, sanitizers and sinks in initial specification versus the inferred specification tested on 25 random reported specifications

Reason	Seed spec	Inferred spec
True vulnerabilities	24%	28%
Vulnerable flow, but no bug	28%	12%
Incorrect sink	0%	24%
Incorrect source	0%	8%
Incorrect source and sink	0%	8%
Missing sanitizer	40%	8%
Flows into wrong parameter	8%	12%

Q3: Number of Sources, Sanitizers, and Sinks. With the above threshold, Seldon identified numerous sources, sanitizers, and sinks, shown in Tab. 5. Identifying a similarly large set by hand would be difficult as only a small fraction of all candidates (3.27%) have some role. Even inspecting all inferred specifications manually is significantly cheaper than inspecting all candidates, by a factor of about 30.

Our results are in line with those in [17], which identified 3.8% of all nodes in its propagation graph as sources, sanitizers, or sinks (381 out of 10038). However, in their setting, the learning problem was significantly easier, as C# allows filtering candidates by type, thus removing 75% of all nodes in the propagation graph.

Q4: Number of Vulnerabilities. To evaluate the usefulness of inferred specifications for detecting security vulnerabilities in real web applications, we ran taint analysis on our complete GitHub dataset (described in §7.2) using both the

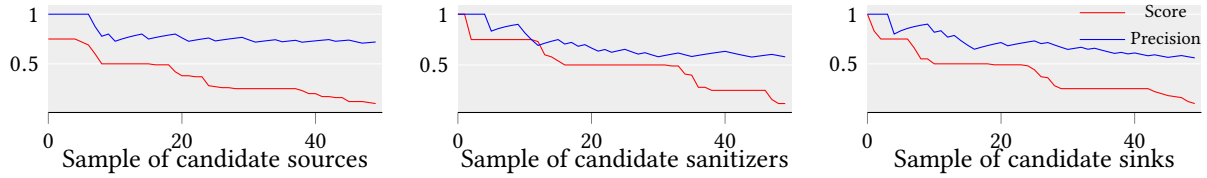


Figure 11. Plot of 50 candidate samples for sources, sanitizers and sinks. The samples are sorted according to their predicted score for the role. Precision for all samples up to a given sample shows how higher score correlates with precision.

Table 7. Total number of reports and estimated number of vulnerabilities in all the data based on evaluating true positive rate on the sample from Tab. 6

Reason	Seed spec	Inferred spec
Number of reports	662	21318
Number of projects affected	192	2409
Estimated vulnerabilities	159	5969

initial seed specifications and the final inferred specifications. In each case we manually inspected 25 randomly sampled vulnerability reports. The comparison is shown in Tab. 6.

In terms of precision as estimated on the 25 samples, both approaches discover similar ratio of true vulnerable flows. However, not all of these flows are exploitable due to other settings in the evaluated applications. For example, a Cross-Site Scripting attack is not possible if the `Content-Type` http header is set to `text/plain` and not `text/html`. Another observation is that the seed specification misses a number of sanitizers and produces false positives for sanitized flows. In contrast, the inferred specification can contain incorrect sources and sinks, but has increased coverage for sanitizers.

In terms of recall, the results are summarized in Tab. 7. The seed specification only produces a small number of total reports (662), but these reports still contain a large number of false positives. The inferred specification significantly increases the number of reports to 21318 in 2409 projects and we estimate the total number of true security vulnerabilities detected by Seldon to be close to 6000.

Based on this sample, we conclude that the learned specifications increase the estimated number of true vulnerabilities in our dataset by an order of magnitude.

Q5: Impact of Large Dataset. Next, we evaluated the impact of learning on a big dataset as opposed to on a single project. To this end, we randomly selected 3 projects from the complete corpus (described in Section §7.2) and ran the end-to-end process for each of them separately, always using the same seed specifications. To compare the results to the complete specifications (obtained by learning on the full dataset), we projected the complete specifications to each of the three projects (*i.e.,* we ignored all specifications not occurring in the considered project). For each of the three projects, we then compared the specifications obtained from training only on that project vs training on all projects.

The experiment shows the precision increased from 45% (on average, when using the individual specifications) to 65% (on average, when using the projections of complete specifications). In addition to increasing precision, the complete specification inferred 18 new, true roles not predicted by the individual specifications. We thus conclude that cross-project learning is beneficial.

Q6: Impact of Seed Specification. We also evaluated Seldon's precision for half the seed specification (considering only odd line numbers in App. B). This significantly reduces precision, by 14 percentage-points. Thus, we believe our seed specification strikes a good balance between manual effort and precision. We note that in the extreme case of an empty seed specification, Seldon will predict 0 specifications, because picking 0 for all variables is a trivial solution to its constraint system.

Q7: Reporting Bugs. Finally, to ensure reported vulnerabilities have value in real-world projects, we inspected several reports with highly scored sources and sinks, built proof-of-concept exploits and reported 49 severe vulnerabilities in 17 projects: 25 Cross-Site Scripting, 18 SQL Injections, 3 Path Traversal, 2 Command Injections, and 1 Code Injection. We provide a full list of all reported bugs in App. C. We note that only 3 of these could be discovered using only the seed specification, and that some of the reported issues can affect a large number of users.

8 Related Work

In this section, we describe the work most related to ours.

Specification Mining. Multiple works have proposed learning specifications related to information flow. The work most related to ours is Merlin [17], which we have discussed in detail (§6) and also investigated in our evaluation (§7.4).

SuSi [24] trains a support vector machine (SVM) classifier to identify privacy specifications of Android APIs. In contrast to our work, SuSi is fully supervised and heavily relies on the similarity of type signatures of methods with similar role. While effective for Android, this method is less applicable to the diverse and dynamic Python codebases. In general, Python is more difficult to analyze than statically

typed languages such as Java and C# (targeted by SuSi and Merlin, respectively). Concretely, in contrast to Java and C#, in Python one cannot identify a unique target of an event, filter source, sink, or sanitizer candidates by their type, or leverage statically precise information flow analysis.

Shar et al. [25] predict sanitizers using backward program slicing from a known set of sinks and dynamically computes a number of features for each function (*e.g.,* testing how functions react on dot-dot-slash to sanitize path traversal [21]). Based on this information, their tool predicts if an information flow is sanitized (however it still needs manually provided sources and sinks). Modelgen [4] learns information flow of Android privacy APIs based on dynamic analysis obtained from a large test case set and mined on invocation subtraces. In contrast, our approach is fully static and also works on a range of APIs not covered by test suites.

Taint Analysis. Numerous works have addressed *static* taint analysis in Android, including SCanDroid [11], ScanDal [14], LeakMiner [30], FlowDroid [2] and DroidSafe [12]. Taint analysis has also been investigated for server-side web application [27], for client-side JavaScript [28] or for C [29].

Other works have applied *dynamic* taint analysis for a range of applications such as TaintDroid [8], DTA++ [13], Python taint for Erasure [5] and TaintCheck for Valgrind [20].

All these static and dynamic approaches require a specification of sources, sanitizers, and sinks, based on which they perform taint analysis. Therefore, the work on these tools is complementary to our approach: Better specifications improve these tools, while better information flow analysis may improve our specification learning.

9 Conclusion and Discussion

We presented Seldon, a new approach for learning taint specifications. Its key idea is to phrase the task of inferring the specification as a linear optimization, leading to scalable inference and enabling simultaneous learning of specifications over a large dataset of programs.

Clearly, Seldon is subject to the usual limitations of taint analysis: it can only help to detect vulnerabilities based on information flow and does not address, *e.g.,* buffer overflows or insecure configurations. In addition, Seldon relies on a high-quality dataset and on a correctly annotated (but small) seed specification. For example, when the dataset often misses sanitization, the constraint from Fig. 4c forces the variables to (i) omit the specification of the source or sink, (ii) incorrectly label a non-sanitizer as a sanitizer, or (iii) accept a penalty for this constraint.

In our evaluation, we showed that Seldon can infer thousands of taint specifications for Python programs with high estimated precision. Manual examination of the reports produced by our taint analyzer (which uses the inferred specifications) indicates the reports contain true security violations, some of which we reported to the maintainers of the projects.

Acknowledgements

The research leading to these results was partially supported by an ERC Starting Grant 680358.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[3] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, Jeffrey Dean, and Google Inc. 2007. Large language models in machine translation. In *In EMNLP*. 858–867.

[4] Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: Mining Explicit Information Flow Specifications from Concrete Executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/2771783.2771810

[5] Juan José Conti and Alejandro Russo. 2012. A Taint Mode for Python via a Library. In *Proceedings of the 15th Nordic Conference on Information Security Technology for Applications (NordSec'10)*. Springer-Verlag, Berlin, Heidelberg, 210–222. https://doi.org/10.1007/978-3-642-27937-9_15

[6] Gregory F. Cooper. 1990. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence* 42, 2-3 (March 1990), 393–405. https://doi.org/10.1016/0004-3702(90)90060-D

[7] Paul Dagum and Michael Luby. 1993. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence* 60, 1 (March 1993), 141–153. https://doi.org/10.1016/0004-3702(93)90036-B

[8] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems* 32, 2 (June 2014), 1–29. https://doi.org/10.1145/2619091

[9] Flask Admin. [n. d.]. https://github.com/flask-admin/flask-admin/.

[10] Flask API. [n. d.]. www.flaskapi.org.

[11] Adam Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. SCanDroid: Automated security certification of Android applications. (01 2009).

[12] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-Flow Analysis of Android Applications in DroidSafe. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2015.23089

[13] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Xiaodong Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*.

[14] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. 2018. SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. (11 2018).

[15] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[16] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press.

[17] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. *SIGPLAN Not.* 44, 6 (June 2009), 75–86. https://doi.org/10.1145/1543135.1542485

[18] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251398.1251416

[19] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. /Infer.NET 0.3. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[20] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. http://www.isoc.org/isoc/conferences/ndss/05/proceedings/papers/taintcheck.pdf

[21] OWASP. 2015. OWASP on Path Traversal. https://www.owasp.org/index.php/Path_Traversal.

[22] OWASP. 2017. OWASP Top 10 Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[23] Python Taint. [n. d.]. https://github.com/python-security/pyt.

[24] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings 2014 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2014.23039

[25] L. K. Shar, L. C. Briand, and H. B. K. Tan. 2015. Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning. *IEEE Transactions on Dependable and Secure Computing* 12, 6 (Nov 2015), 688–707. https://doi.org/10.1109/TDSC.2014.2373377

[26] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. https://doi.org/10.1561/2500000014

[27] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 210–225. https://doi.org/10.1007/978-3-642-37057-1_15

[28] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM SIGPLAN Notices* 44, 6 (May 2009), 87. https://doi.org/10.1145/1543135.1542486

[29] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 797–812. https://doi.org/10.1109/SP.2015.54

[30] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. IEEE, Wuhan, China, 101–104. https://doi.org/10.1109/WCSE.2012.26

[31] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. 2003. Exploring Artificial Intelligence in the New Millennium. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 239–269. http://dl.acm.org/citation.cfm?id=779343.779352

API	Score	Correct
post.title	0.75	✓
claimed_user.fullname	0.75	✓
create_site_element(param siteResult).Comments	0.75	✓
filen.filename	0.75	✓
file_instance.filename	0.75	✓
flask.views.MethodView::get(param filename)	0.72	✓
flask.request.form['srpValueM']	0.69	✓
anyaudio.helpers.encryption.decode_data()['url']	0.6	
urlparse.urlparse().port	0.5	
create(param name)	0.5	✓
_create_or_edit(param entry).slug	0.5	
LoginForm().username.data	0.5	✓
edit(param name)	0.5	✓
feed_get(param handle)	0.5	✓
flask.request.form['name']	0.5	✓
flask.g.site.publish_log_file	0.5	
inputtemplate.filename	0.49	✓
get_function_data(param jid)	0.49	✓
_process_new_comment(param comment)	0.49	✓
u.username	0.42	✓
lava_results_app.models.QueryCondition.get_similar_job_content_types()	0.38	
response_add(param obj).pk	0.38	
playlist_edit(param request).user	0.37	✓
pymongo.Connection().wbuserstatus.collection_names()	0.37	✓
flask.request.form['json']	0.28	✓
rpt.request().bound_action.action.default_format	0.27	
edit(param path)	0.26	✓
service_unavailable_handler(param error).description[1]	0.26	✓
flask.request.form['bookmark_path']	0.25	✓
form.GeneralForm().username.data	0.25	✓
_hash()[]	0.25	
execute(param context)	0.25	
flaskup.models.SharedFile().get()	0.25	✓
self.request	0.25	✓
quota_edit(param request)	0.25	✓
render_html(param template)	0.25	
robots(param request)	0.25	✓
args.REQUEST['body']	0.25	✓
google.appengine.ext.webapp.template.django.template.Node::render(param self).src_key	0.23	
djangoffice.forms.users.UserForm().cleaned_data['email']	0.2	✓
resp['user_id']	0.2	✓
bathymetry(param projection)	0.17	✓
models.User().get()	0.17	✓
flask.current_app.root_path	0.16	
upload_photo.filename	0.16	✓
poll_details(param hackathon_name)	0.12	✓
flask.request.user.id	0.12	
argparse.ArgumentParser().parse_args().port	0.12	
flask.request.stream.read()	0.11	✓
flask.request.args.get()[6]	0.1	✓

Table 8. Evaluation on 50 random events classified as sources by Seldon.

A Evaluated Samples

Tables 8, 9 and 10 include detailed evaluation for the random sample of sources, sanitizers and sinks predicted by Seldon. In general, it is quite challenging to even check these specifications, if they are not shown in the context of the flows they appear in, let alone coming up with them directly.

API	Score	Correct
flux.models.User().url()	1	✓
hunchworks.forms.EvidenceForm().save()	1	✓
clam.common.util.xmlescape()	0.75	✓
common.twitter.get_signin_url()	0.75	✓
form_valid(param self).request.cradmin_app.reverse_appurl()	0.75	✓
shiva.models.Album()	0.75	
utils.helper.MethodView::post(param repo).meta.get_blob()	0.75	✓
imgur.getAuthUrl()	0.75	✓
flask.current_app.aws.connect_to().generate_url()	0.75	✓
util.jwt_encode()	0.75	✓
sahara.utils.wsgi.JSONDictSerializer().serialize()	0.75	
vm_templates.models.VirtualMachineTemplate.objects.get()	0.75	
redwind.controllers.process_people()	0.73	
dci.server.auth.hash_password()	0.6	✓
QuickEntryForm().save()	0.58	✓
markdown.Markdown().convert()	0.54	✓
helpers.normalize()	0.5	
ba.action.get_target_url()	0.5	✓
ghdata.GHTorrent().contributions()	0.5	
transformations.hourly()	0.5	✓
mediamanager.MediaManager.cover_art_uuid()	0.5	
opencricket.chart.syntax_response.SyntaxResponse.build_response()	0.5	
box.get_authorization_url()	0.5	✓
urllib.parse.urlunparse()	0.5	
silopub.util.set_query_params()	0.5	✓
storage.load().content_redirect_url()	0.5	✓
faitoutlib.get_new_connection()	0.5	
files.models.File.from_upload()	0.5	
PayOrderByPaypal.get_paypal_url()	0.5	✓
viz_obj.json_dumps()	0.5	
course.utils.get_flow_access_rules()	0.5	
flask_cas.cas_urls.create_cas_logout_url()	0.5	✓
werkzeug.secure_filename()	0.49	✓
urllib.urlencode()	0.49	✓
generateSavegame.createZip()	0.41	
jsonConverter.parse_object()	0.4	
portality.dao.Facetview2.url_encode_query()	0.28	✓
core.obfuscate()	0.28	✓
octavia.amphorae.backends.agent.api_server.util.haproxy_dir()	0.25	✓
util.shorten()	0.25	✓
des.encrypt()	0.25	✓
dpxdt.server.models.Artifact()	0.25	
gzip.open()	0.25	
next().process()	0.25	
esp.program.models.TeacherBio.getLastBio()	0.25	
auth.generate_state()	0.25	✓
RentalManager()	0.25	✓
baiducloudengine.BaiduCloudEngine().check_file()	0.16	✓
markdown()	0.12	
onlineforms.models.FormFiller.objects.create()	0.12	

Table 9. Evaluation on 50 random events classified as sanitizers by Seldon.

API	Score	Correct
fs.write()	1	✓
subprocess.Popen().communicate()	0.83	✓
common.mail.SendEmail().send()	0.75	✓
utils.account.send_verify_mail()	0.75	✓
ShellCommandError()	0.75	
mail.send_mail()	0.75	✓
CommentPostBadRequest()	0.75	✓
flask.Blueprint().add_url_rule()	0.66	✓
django.utils.six.with_metaclass()::error_response(param response_class)()	0.55	✓
flask.helpers.make_response()	0.55	✓
catsnap.resize_image.ResizeImage.make_resizes()	0.5	
ext.render_template()	0.5	
flask_mail.Mail().send()	0.5	✓
flaskext.sqlalchemy.SQLAlchemy().session.add()	0.5	
db.session.add()	0.5	
User.query.filter_by()	0.5	
flask.Markup()	0.5	✓
django.core.mail.EmailMessage().attach()	0.5	✓
fp_new.write()	0.49	✓
db.write()	0.49	✓
models.Gcm.send_message()	0.49	
flask.current_app.mail.send()	0.49	✓
open().write()	0.49	✓
requests.post()	0.48	✓
self.response.out.write()	0.44	✓
PyPDF2.PdfFileReader()	0.37	
CookieForm().prepareResponse()	0.36	✓
qs[0].set_status()	0.28	
groups.models.all_activities()	0.25	
lastuser_core.models.db.session.add()	0.25	
common.messaging.models.DatedMessage.objects.post_message_to_user()	0.25	✓
baseframe.forms.render_redirect()	0.25	✓
textpress.views.admin.render_admin_response()	0.25	
upload(param self).container.upload_object_via_stream()	0.25	✓
blueprints.admin.models.Users.check_user_passwd()	0.25	
dismod3.plotting.plot_posterior_region()	0.25	
object_log.models.LogItem.objects.log_action()	0.25	
flask.abort()	0.25	✓
editors.helpers.ReviewHelper().set_data()	0.25	
textpress.views.admin.flash()	0.25	✓
fumblerooski.utils.calculate_record()	0.25	
redwind.models.Post.load_by_path()	0.25	
self._fileHandler.addNewFile()	0.22	✓
os.dup2()	0.2	
hooks.fire()	0.18	
cursor.execute()	0.17	✓
anarcho.apk_helper.parse_apk()	0.16	✓
standardweb.models.Message.query.options().filter_by()	0.12	
db.session.delete()	0.1	

Table 10. Evaluation on 50 random events classified as sinks by Seldon.

B Initial Seed Specifications

Below we provide as a listing our initial seed specification. Lines starting with `o:` denote sources, with `a:` sanitizers, with `i:` sinks and with `b:` blacklisted events from these roles:

```
# Sources
o: User.objects.get()
o: cms.apps.pages.models.Page.objects.get()
o: django.core.extensions.get_object_or_404()
o: django.http.QueryDict()
o: django.shortcuts.get_object_or_404()
o: example.util.models.Link.objects.get()
o: flask.request.form.get()
o: inviteme.forms.ContactMailForm()
o: live_support.forms.ChatMessageForm()
o: model_class.objects.get()
o: req.form.get()
o: request.GET.copy()
o: request.GET.get()
o: request.POST.copy()
o: request.POST.get()
o: request.args.get()
o: request.form.get()
o: request.pages.get()
o: self.get_query_string()
o: self.get_user_or_404()
o: self.queryset().get()
o: self.request.FILES.get()
o: self.request.get()
o: self.request.headers.get()
o: textpress.models.Page.objects.get()
o: textpress.models.Tag.objects.get()
o: textpress.models.User()
o: textpress.models.User.objects.get()

# SQL injection
i: MySQLdb.connect().cursor().execute()
i: MySQLdb.connect().execute()
a: MySQLdb.connect().cursor().mogrify()
a: MySQLdb.escape_string()

i: pymysql.connect().cursor().execute()
i: pymysql.connect().execute()
a: pymysql.connect().cursor().mogrify()
a: pymysql.escape_string()

i: pyPgSQL.connect().cursor().execute()
i: pyPgSQL.connect().execute()
a: pyPgSQL.connect().cursor().mogrify()
a: pyPgSQL.escape_string()

i: psycopg2.connect().cursor().execute()
i: psycopg2.connect().execute()
a: psycopg2.connect().cursor().mogrify()
a: psycopg2.escape_string()

i: sqlite3.connect().cursor().execute()
i: sqlite3.connect().execute()
a: sqlite3.connect().cursor().mogrify()
a: sqlite3.escape_string()

i: flask.SQLAlchemy().session.execute()
i: SQLAlchemy().session.execute()
i: db.session().execute()
i: flask.SQLAlchemy().engine.execute()
i: SQLAlchemy().engine.execute()
i: db.engine.execute()

i: django.db.models.Model::objects.raw()
i: django.db.models.expressions.RawSQL()
i: django.db.connection.cursor().execute()

# XPath Injection
i: lxml.html.fromstring().xpath()
i: lxml.etree.fromstring().xpath()
i: lxml.etree.HTML().xpath()

# OS Command Injection
i: subprocess.call()
i: subprocess.check_call()
i: subprocess.check_output()
i: os.system()
i: os.spawn()
i: os.popen()

a: subprocess.Popen()

# XXE
i: lxml.etree.to_string()

# XSS
i: amo.utils.send_mail_jinja()
i: django.utils.html.mark_safe()
i: django.utils.safestring.mark_safe()
i: example.util.response.Response()
i: jinja2.Markup()
i: olympia.amo.utils.send_mail_jinja()
i: suds.sax.text.Raw()
i: swift.common.swob.Response()
i: webob.Response()
i: wtforms.widgets.HTMLString()
i: wtforms.widgets.core.HTMLString()
i: flask.Response()
i: flask.make_response()
i: flask.render_template_string()

a: bleach.clean()
a: cgi.escape()
```

```
a: django.forms.util.flatatt()
a: django.template.defaultfilters.escape()
a: django.utils.html.escape()
a: flask.escape()
a: jinja2.escape()
a: textpress.utils.escape()
a: werkzeug.escape()
a: werkzeug.html.input()
a: xml.sax.saxutils.escape()
a: flask.render_template()
a: django.shortcuts.render()
a: django.shortcuts.render_to_response()
a: django.template.Template().render()
a: django.template.loader.get_template().render()
a: werkzeug.exceptions.BadRequest()

# Path Traversal
i: flask.send_from_directory()
i: flask.send_file()

a: os.path.basename()
a: werkzeug.utils.secure_filename()

# Open Redirect
i: flask.redirect()
i: django.shortcuts.redirect()
i: django.http.HttpResponseRedirect()

# Black list

# Imports and related functions.
b: *tensorflow*
b: *tf*
b: *numpy*
b: *pandas*
b: np.*
b: plt.*
b: pyplot.*
b: os.path.*
b: uuid.*
b: sys.*
b: json.*
b: datetime.*
b: io.*
b: re.*
b: hashlib.*
b: struct.*
b: *String*
b: *Queue*
b: threading*
b: mutex*
b: dummy_threading*
b: multiprocessing*
b: *module*
b: math.*

# Flask
b: flask.Flask()*
b: app.*

# Django
b: *django*conf*
b: *django*settings*
b: *ugettext*
b: *lazy*
b: *RequestContext*

# Logs
b: *logging*
b: *logger*

b: tempfile.mkdtemp()
b: type().__name__
b: set_size(param n)
b: result.append()
b: str().encode()
b: ValueError()
b: logging.info()
b: key.split()
b: json.dump()

# Python built-ins.
b: False
b: None
b: True
b: *_()*
b: __import__()
b: *__name__*
b: *_str()*
b: *_unicode()*
b: abs()
b: *.all()
b: *.any()
b: *.append()
b: ascii()
b: *assert*
b: attr()
b: bin()
b: bool()
b: builtins.str()
b: bytearray()
b: bytes()
b: *.capitalize()
b: *.center()
b: chr()
b: classmethod()
b: cmp()
b: complex()
```

```
b: *.copy()
b: *.count()
b: *.decode()
b: dict()
b: *.difference()
b: *.difference_update()
b: dir()
b: *.encode()
b: *.endswith()
b: enumerate()
b: *.extend()
b: *.filter()
b: *.find()
b: *.findall()
b: *.finditer()
b: float()
b: *.format()
b: frozenset()
b: func()
b: future.builtins.str()
b: getattr()
b: globals()
b: hasattr()
b: hash()
b: help()
b: hex()
b: id()
b: *.index()
b: *.insert()
b: int()
b: *.intersection()
b: *.intersection_update()
b: *.isalnum()
b: *.isalpha()
b: *.isdecimal()
b: *.isdigit()
b: *.isdisjoint()
b: *.isidentifier()
b: *.isinstance()
b: *.islower()
b: *.isnumeric()
b: *.isprintable()
b: *.isspace()
b: *.issubclass()
b: *.issubset()
b: *.issuperset()
b: *.istitle()
b: *.isupper()
b: *.keys()
b: kwargs
b: *len()
b: list()
b: *.ljust()
b: locals()
b: *.lower()
b: *.lstrip()
b: *.maketrans()
b: *.map()
b: *.match()
b: *.match.group()
b: max()
b: meth()
b: min()
b: next()
b: object()
b: oct()
b: open()
b: ord()
b: *.pop()
b: *.popitem()
b: pow()
b: print()
b: *.purge()
b: *.quote()
b: *.quoted_url()
b: range()
b: reduce()
b: *.reload()
b: *.remove()
b: *.replace()*
b: *.repr()
b: *.reverse()
b: reversed()
b: *.rfind()
b: *.rindex()
b: *.rjust()
b: round()
b: *.rpartition()
b: *.rsplit()
b: *.rstrip()
b: *.search()
b: set()
b: setattr()
b: *.setdefault()
b: *.sort()
b: sorted()
b: *.split()*
b: *.splitlines()
b: *.startswith()
b: *.staticmethod()
b: str
b: str()
b: *.strip()
b: strip_date.strftime()
b: *.sub()
b: *.subn()
b: sum()
b: super()
```

Pull Request	Number of Bugs	Type of Bug
https://github.com/anyaudio/anyaudio-server/pull/163	2	XSS
https://github.com/DataViva/dataviva-site/issues/1661	2	Path Traversal
https://github.com/DataViva/dataviva-site/issues/1662	1	XSS
https://github.com/earthgecko/skyline/issues/85	1	XSS
https://github.com/earthgecko/skyline/issues/86	2	SQLi
https://github.com/gestorpsi/gestorpsi/pull/75	2	XSS
https://github.com/HarshShah1997/Shopping-Cart/pull/2	12	SQLi
https://github.com/kylewm/silo.pub/issues/57	1	XSS
https://github.com/kylewm/woodwind/issues/77	2	XSS
https://github.com/LMFDB/lmfdb/pull/2695	7	XSS
https://github.com/LMFDB/lmfdb/pull/2696	1	SQLi
https://github.com/mgymrek/pybamview/issues/52	1	Command Injection
https://github.com/MinnPost/election-night-api/issues/1	1	Command Injection
https://github.com/mitre/multiscanner/issues/159	1	Path Traversal
https://github.com/MLTSHP/mltshp/pull/509	1	XSS
https://github.com/mozilla/pontoon/pull/1175	5	XSS
https://github.com/PadamSethia/shorty/pull/4	1	SQLi
https://github.com/sharadbhat/VideoHub/issues/3	1	SQLi
https://github.com/UDST/urbansim/issues/213	1	Code Injection
https://github.com/viaict/viaduct/pull/5	3	XSS
https://github.com/yashbidasaria/Harry-s-List-Friends/issues/1	1	SQLi

Table 11. Reported Bugs.

```
b: *.symmetric_difference()
b: *.symmetric_difference_update()
b: *test*
b: *.translate()
b: *.trim_url()
b: *.truncate()
b: tuple()
b: *.type()
b: unichr()
b: unicode()
b: unknown()
b: *.update()
b: *.upper()
b: *.values()
b: *.vars()
b: zip()
```

C Reported Bugs

In Tab. 11, we provide a list of all pull requests based on bugs reported by our taint analysis.