Hash-Flow Taint Analysis of Higher-Order Programs

Shuying Liang  Matthew Might
University of Utah
{liangsy,might}@cs.utah.edu

Abstract
As web applications have grown in popularity, so have attacks on such applications. Cross-site scripting and injection attacks have become particularly problematic. Both vulnerabilities stem, at their core, from improper sanitization of user input.

We propose static taint analysis, which can verify the absence of unsanitized input errors at compile-time. Unfortunately, precise static analysis of modern scripting languages like Python is challenging: higher-orderness and complex control-flow collide with opaque, dynamic data structures like hash maps and objects. The interdependence of data-flow and control-flow make it hard to attain both soundness and precision.

In this work, we apply abstract interpretation to sound and precise taint-style static analysis of scripting languages. We first define $\lambda_H$, a core calculus of modern scripting languages, with hash maps, dynamic objects, higher-order functions and first class control. Then we derive a framework of $k$-CFA-like CESK-style abstract machines for statically reasoning about $\lambda_H$, but with hash maps factored into a “Curried Object store.” The Curried object store—and shape analysis on this store—allows us to recover field sensitivity, even in the presence of dynamically modified fields. Lastly, atop this framework, we devise a taint-flow analysis, leveraging its field-sensitive, interprocedural and context-sensitive properties to soundly and precisely detect security vulnerabilities, like XSS attacks in web applications.

We have prototyped the analytical framework for Python, and conducted preliminary experiments with web applications. A low rate of false alarms demonstrates the promise of this approach.

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

General Terms Languages, Security

Keywords Static Analysis, Taint Analysis, Abstract Interpretation, Higher-order programs

1. Introduction
Web applications have exploded in popularity over the past decade. Security risks for web applications have risen in tandem [2]. According to OWASP [24], XSS (cross-site scripting) and SQL injection have been the top two most common security threats for several years. Both are caused by unsanitized user input flowing into unsafe or privileged sinks such as system calls, database queries or, in modern times, client-side HTML.

Taint analysis has been proposed to solve this problem [10, 30], by tracking and detecting whether tainted (unsanitized user input) values may flow into security sinks. To date, the dominant methods for taint analysis are dynamic [12, 13, 23, 26, 29, 33]. The main strength of the dynamic approach is that it can track tainted information through direct data dependencies in an efficient and precise way [13]. However, strictly dynamic analyses cannot prevent soft failure of the application when an unexpected taint violation happens [5, 27, 32]. The strength of static taint analyses is that they can prove the absence of taint violations, or at least delimit the regions in which they may happen.

However, sound and precise static analysis of modern scripting languages is hard. The origin of the analysis problem is the intersection of dynamism and higher-orderness. Specifically, there are four interacting facets of dynamic scripting languages: (1) dynamic allocation and modification of hash maps; (2) higher-order functions and first-class control; (3) dynamic computation and manipulation of keys; and (4) reflective access and modification of hash maps and objects, making control-flow analysis and data-flow analysis inseparable from one another and thus complicating static analysis.

Figure 1 contains a simple, cut-down example of how hash tables and higher-orderness can interact in dynamic languages.

Existing analyses often don’t consider these interactions, or they attempt to model hash maps as objects, a strategy which collapses to field-insensitivity reasoning when pricked with even the slightest imprecision.

We provide an extensible analytic framework rooted in abstract machines for weaving analysis of the four facets together. The framework supports generalization of higher-order shape-analytic techniques [17], and fuses them with rich abstract domains for strings to create analytic techniques for hash maps. We then augment this sound, precise analytic framework with static taint analysis.

1.1 Overview
In this work, we provide a systematic approach to the static taint analysis for web programs written in scripting languages—even in the presence of dynamic hash maps, higher-order functions and first-class control. The ability to analyze these features in tandem is
function processCommand(cmd) {
  handlers["before_MSG"] =
    function (cmd) { /* run before handling MSG */ };
  handlers["after_MSG"] =
    function (cmd) { /* run after handling MSG */ };
  handlers["before_PRIVMSG"] =
    function (cmd) { /* run before handling PRIVMSG */ };
  handlers["after_PRIVMSG"] =
    function (cmd) { /* run after handling PRIVMSG */ };
  // ...
  function cmd.process() {
    handlers["before_" + cmd.action](cmd);
    handlers["after_" + cmd.action](cmd);
  }
}

Figure 1. An example illustrating the interplay of hash tables and higher-orderness in dynamic scripting languages. Without precise reasoning about the structure of the hash table, the calls to the handler table will be seen as invoking all functions inside the table.

By arranging a critical step towards a practical static analyzer not just for static taint analysis, but for deep static analysis of these languages in general.

We design $\lambda_H$, which is a core calculus of modern scripting languages, with hash maps, dynamic objects, higher-order functions and first-class control. Then we devise a framework of k-CFA-like CESK-style abstract machines for statically reasoning about $\lambda_H$, but with the abstract domain of hash maps factored into a “Curried object store.” This factoring allows us to recover field-sensitivity in the presence of dynamically modified fields. Based on the resulting framework, we produce a taint analysis, leveraging field-sensitivity, interprocedural control-flow, and context-sensitivity, to handle both explicit and implicit information flow (by default), and to detect security vulnerabilities, such as XSS attacks, in web applications.

Briefly, we make the following contributions:

- $\lambda_H$: A core calculus of modern scripting languages.
- A “hash-flow analysis” framework for analyzing dynamic objects, hash maps with dynamic keys, first-class control and higher-order functions.
- Cardinality-based shape analysis on hash maps with dynamic keys, in order to support strong update (and deletion) of both values and taint information.
- HFTA: A static “hash-flow taint analysis” based on our sound analytic framework.

2. The setting: $\lambda_H$

Figure 2 presents the syntax of $\lambda_H$—a core calculus for modern dynamic scripting languages. It models mutable hash maps; classes and objects (which are desugared into maps); higher-orderness; first-class control and dynamic string manipulation.

$\lambda_H$ contains the core ingredients necessary to observe collisions between the four facets: integer and string constants; first-class continuations to subsume constructs like exceptions; first-class-closures; simple primitives for operating on integers and strings; mutable hash maps; and a form for introspectively iterating over hash maps.

Syntactically and semantically, strings are sequences of characters:

$s \in String = C^*$

To simplify analysis, the syntax for $\lambda_H$ is A-Normalized [8], but every development in this work could be replayed for a full direct-style language by mildly enriching the structure of continuations to account for the added local evaluation contexts.

3. Analysis design

To motivate the structure of our upcoming analytic framework, we start with a straightforward instrumentation of a traditional concrete semantics to perform static taint analysis. We then attempt a direct abstract interpretation of this instrumented semantics. The resulting analyzer exhibits several shortcomings in its reasoning about objects with dynamically modified field names, and fixing these shortcomings leads to both the Curried object store and a new analytic architecture (Section 4).

To design a taint analysis, one could extend an existing CESK-style [6, 7] abstract machine for $\lambda_H$ by adding an additional taint store, as presented in Figure 3.

Each machine state contains the current expression, the current environment, the store, an extended taint store, the current continuation, and a time-stamp. Environments map variables to addresses; stores map addresses to values; taint stores map addresses to the taintedness of that address.

Time-stamps are required to strictly increase each time the machine steps forward, so they can serve as a source of freshness for al-
lifts abstraction element-wise across tuples:
\[ \alpha(x_1, \ldots, x_n) = \langle \alpha(x_1), \ldots, \alpha(x_n) \rangle, \]
monotonically and point-wise across functions:
\[ \alpha(f) = \lambda x. \bigwedge_{\alpha(s) \leq \alpha(t)} \alpha(f(x)), \]
and member-wise across sets:
\[ \alpha \{ x_1, \ldots, x_n \} = \bigsqcup_i \{ \alpha(x_i) \}. \]

To fully define a structural abstraction, one needs to define only abstractions over the “leaves” of a concrete state-space. In the case of \( \lambda_H \), these leaves are strings (\( \text{String} \)), integers (\( \mathbb{Z} \)), times (\( \text{Time} \)) and object/hash-map locations (\( \text{OLoc} \)). We use Galois connections to abstract them as follows:

**Strings (\( \text{String} \))** For now, we leave the abstract domain for strings opaque as the lattice \( \text{String} \). That is, we can assume that there exists a Galois connection between sets of concrete strings and abstract strings:
\[ (\mathcal{P}(\text{String}), \subseteq) \xrightarrow{\gamma} (\hat{\text{String}}, \sqsubseteq). \]

This Galois connection is effectively a parameter for analysis designers to tune. Some candidates for this Galois connection include a flat constant lattice, the lattice of regular languages, and lattices induced by context-free grammars.

**Integers (\( \mathbb{Z} \))** We assume a similar Galois connection for integers:
\[ (\mathcal{P}(\mathbb{Z}), \subseteq) \xrightarrow{\gamma} (\hat{\mathbb{Z}}, \sqsubseteq). \]

One might use, for instance, the domain of signs or intervals.

**Time (\( \text{Time} \))** We can assume that the analysis designer is supplying an abstraction on times:
\[ (\mathcal{P}(\text{Time}), \subseteq) \xrightarrow{\alpha} (\hat{\text{Time}}_\bot, \sqsubseteq). \]

This Galois connection drives the context-sensitivity of the analysis.

**Object/Hash-map Locations (\( \text{OLoc} \))** We assume a similar Galois connection for object locations:
\[ (\mathcal{P}(\text{OLoc}), \subseteq) \xrightarrow{\alpha} (\hat{\text{OLoc}}_\bot, \sqsubseteq). \]

The structure of this Galois connection drives the polyvariance of the analysis with respect to objects.

### 3.2 Problem 1: A partially ordered abstract address space

When lifting the abstraction to stores (including the traditional store as well as the extended taint store), awkwardness creeps in as addresses acquire a true partial order from abstract values. Originally, an abstract store maps a flat domain (the set of addresses) into a partially ordered codomain (the set of abstract denotable values). Unfortunately, the abstract addresses that result from structural abstraction have a partial order to them.

The structural abstraction of addresses is:
\[ \alpha(\text{bindaddr}(v, t)) = \text{bindaddr}(v, \alpha \{ t \}) \]
\[ \alpha(\text{fieldaddr}(\ell, s)) = \text{fieldaddr}(\alpha \{ \ell \}, \alpha \{ s \}). \]

which yields the following abstract address space:
\[ \hat{a} \in \hat{\text{Addr}} := \text{bindaddr}(v, t) \]
\[ \tag{\hat{f}} \text{fieldaddr}(\hat{\ell}, \hat{s}). \]
Because String is not a flat lattice, addresses are not flat either, which means that both abstract store and abstract taint store must be monotonic maps:

\[
\text{Store} = \text{Addr} \xrightarrow{\text{mon}} \hat{D} \\
\text{Taint} = \text{Addr} \xrightarrow{\text{mon}} \hat{T},
\]

rather than (traditionally) flat maps. Recall that a function \( f \) is monotonic iff

\[ x \subseteq y \implies f(x) \subseteq f(y). \]

And, join on monotonic functions is point-wise:

\[ (f \sqcup g)(x) = f(x) \sqcup g(x). \]

### 3.3 Ramifications of monotonicity in the abstract store

Consider the following code snippet for Python:

```python
; original Python code:
; o[f1] = 3
; return o[f2]
(let ([t (set! o f1) 3])
  (get o f2))
```

We may find during analysis at this site that the abstract value of variable \( f1 \) is \{foo, bar\}. In other words, we may be setting either the key foo or the key bar. So, the analysis must model both possibilities.

At first glance, the correct action seems simple: if the object \( o \) is at abstract location \( \hat{\ell} \), then we must join the abstract store with a new entry:

\[ \hat{\sigma}' = \hat{\sigma} \sqcup [\text{fieldaddr}(\hat{\ell}, \{\text{foo}, \text{bar}\}) \mapsto \alpha(3)]. \]

But, this is incorrect.

#### 3.3.1 The first twist: Updates are monotonic

If we were to do no more than insert just this entry into the abstract store, it would break the monotonicity of the store. To see why, suppose that the abstract value of variable \( f2 \) is \{foo, bar, qux\}. It is clearly the case that \{foo, bar\} \subseteq \{foo, bar, qux\}, yet the address for the latter is unaffected by the insertion, and therefore fails to account for the new value.

When we insert into a monotonic store, we must insert monotonic maps. We use the ceiling notation for monotonic maps:

\[ [x \mapsto y] = \lambda x'. \begin{cases} y & x \subseteq x' \\ \bot & \text{otherwise.} \end{cases} \]

So, at second glance, it seems that we need to join the abstract store with a monotonic map:

\[ \hat{\sigma}' = \hat{\sigma} \sqcup [\text{fieldaddr}(\hat{\ell}, \{\text{foo}, \text{bar}\}) \mapsto \alpha(3)], \]

into the abstract store.

But, this is also unsound.

#### 3.3.2 The second twist: Updates are also antitonic

If we were to do no more than insert this monotonic map, it would fail to report the proper value on addresses weaker than the inserted address. Consider the case where the value of \( f2 \) is \{foo\}. According to the update, the value of the object at key foo could contain \( \alpha(3) \). But, this value isn’t there.

To remain sound, updates to the the store must be made with respect to submaps of the map to be joined as well. That is, the analysis should perform:

\[ \hat{\sigma}' = \hat{\sigma} \sqcup \left\{ \{\hat{a} \mapsto \alpha(3)\} : \bot \subseteq \hat{a} \subseteq \text{fieldaddr}(\hat{\ell}, \{\text{foo}, \text{bar}\}) \right\}. \]

This is sound.

But, it’s imprecise.

#### 3.3.3 The third twist: Except for bottom

By merging with all submaps of the entry to be added, the analysis adds a map that is monotonic but whose root is bottom. In short, the previous map will join \( \alpha(3) \) to every address in the store.

Fortunately, we can fix this by adjusting the bound to exclude bottom. It is straightforward to prove thissound during the inductive step of a proof of simulation—we need not consider the only case where the key is too precise to exist.

To encapsulate these three twists, we define the “mostly antitonic update operator for monotonic maps”—\( \sqcup\sqcup \):\n
\[ f \sqcup \sqcup [x \mapsto y] = f \sqcup \sqcup \bigcup_{\bot \subseteq x' \subseteq x} [x' \mapsto y]. \]

#### 3.3.4 Moving monotonicity out of the store

Ultimately, performance concerns drive the need to eliminate monotonicity from the store. Look-up and update operations on monotonic maps are not logarithmic like those on flat maps backed by balanced sorted tree maps: the operations are linear, with the expected consequences for algorithmic complexity. In the next section, we will introduce a Curried object store to localize the monotonicity to solely hash maps, thereby isolating the damage to performance and to semantic complexity.

### 3.4 Problem 2: Unbounded continuations

A second apparent problem with the current concrete state-space is that the continuation component is unbounded under a structural abstraction. (Recall that continuations contain continuations.) For classical flow analysis in small-step form, the abstract state-space should be finite. Following Van Horn and Might [31], we can make it finite by threading continuations through the store and then bounding the set of addresses. Since our focus in this work is the structure of the abstract domains for analysis of hash maps, we omit this straightforward refactoring to save space.

### 4. HFTA: Hash Flow Taint Analysis

The previous section revealed shortcomings with respect to the structural abstraction with a standard CESK architecture (as augmented with taint information). The core concern was that order on abstract key values (including strings) leaked a partial order into abstract addresses (via keys in hash maps) which then induced monotonicity in the abstract stores. Monotonicity then mangled the abstract semantics for operations like hash map update, and raised the cost of look-up and update operations on the abstract store from logarithmic to linear.
The major difference from the design illustrated in section 3 is two-fold: Figure 4 presents the concrete state-space for \( \lambda_{HT} \) in Section 4.3. Then we develop abstract semantics in Section 4.2, following that maps within this store to isolate the monotonicity in Section 4.1. We will provide a proof sketch of the soundness of HFTA in Section 4.4. Lastly, we also carve out cardinality shape analysis for HFTA in Section 4.3.

4.1 A concrete semantics for taint analysis

Figure 4 presents the concrete state-space for \( \lambda_{HT} \).

The major difference from the design illustrated in section 3 is two-fold:

- We factored out a separate object store \( OStore \) which is added to each machine state, mapping object locations to objects, and objects are now maps from keys to values. In other words, the structure of the object store is Curried:

  \( OStore : OLoc \rightarrow String \rightarrow D \).

- In addition to factoring \( OStore \) out from the original abstract machine [6, 7], we not only add a taint store \( Taint \) for tracking taint property of ordinary addresses:

  \( Taint : Addr \rightarrow T \),

  but also an object taint store \( OTaint \) to achieve field-sensitivity for the taint property. It resembles the \( OStore \), but with values mapping to taint property space \( T \). That is:

  \( OTaint : OLoc \rightarrow String \rightarrow T \).

The advantage of these two arrangements for object (taint) store becomes apparent under a structural abstraction, where it isolates the monotonicity to individual hash maps:

\[
\begin{align*}
OStore & : OLoc \rightarrow String \xrightarrow{\text{mon}} D \\
O\overline{Taint} & : OLoc \rightarrow String \xrightarrow{\text{mon}} T.
\end{align*}
\]

These refactorings have a second advantage: they support cardinality-like shape analysis on hash maps and also their keys and object fields.

To initiate execution for a program \( pr \), we define the program-to-state injector, \( T : \text{Prog} \rightarrow \text{State} \):

\[
T(pr) = (pr, [], [], [], [], \text{halt}, t_0),
\]

where \( t_0 \) is the initial time.

4.1.1 A concrete transition relation

Next, we need a transition relation on states:

\[
(\Rightarrow) \subseteq \Sigma \times \Sigma.
\]

A straightforward set of rules defines this relation. For instance, \( (\{ [ f \ x_1 \ldots x_n ] \}, \rho, \sigma, \omega, \sigma^T, \omega^T, \kappa, t) \)

\[
\Rightarrow (e, \rho', \sigma', \omega, \sigma^{T'}, \omega^{T'}, \kappa, t'), \quad \text{where} \quad t' = t + 1
\]

\[
((\{ (\lambda \ (v_1 \ldots v_n) \ e) \}, \rho')) = A(f, \rho, \sigma, \omega)
\]

\[
\rho'' = \rho'[v_i \mapsto a_i]
\]

\[
a_i = \text{bindaddr}(v_i, t')
\]

\[
d_i = A(x_i, \rho, \sigma, \omega)
\]

\[
d^{T'}_i = A^T(x_i, \rho, \sigma^T, \omega^T)
\]

\[
\sigma' = \sigma[a_i \mapsto d_i]
\]

\[
\sigma^{T'} = \sigma^T[a_i \mapsto d^{T'}_i],
\]

where \( A : \text{AExp} \times \text{Env} \times \text{Store} \times \text{OStore} \rightarrow D \) is the atomic argument evaluator:

\[
A(c, \rho, \sigma, \omega) = c
\]

\[
A(e, \rho, \sigma, \omega) = \sigma(\rho(e))
\]

\[
A((\lambda \ x \sigma), \rho, \sigma, \omega) = (\lambda \ x, \rho)
\]

\[
A([\text{op} \ d_1 \ldots d_n], \rho, \sigma, \omega) = \text{O}([\text{op}], \sigma, \omega)(d_1, \ldots, d_n)
\]

where \( d_i = A(x_i, \rho, \sigma, \omega) \),

where \( O : \text{Op} \rightarrow \text{Store} \times \text{OStore} \rightarrow D^* \rightarrow D \) is the primitive operation evaluator, and where \( A^T : \text{AExp} \times \text{Env} \times \text{Taint} \times \text{OTaint} \rightarrow T \) is like \( A \), but is evaluating taintedness:

\[
A^T(c, \rho, \sigma^T, \omega^T) = \text{untainted}
\]

\[
A^T((\text{op}, \rho), \sigma^T, \omega^T) = A^T(\text{op}(\rho))(\sigma^T, \omega^T)(d^{T'}_1, \ldots, d^{T'}_n)
\]

where \( d^{T'}_i = A^T(x_i, \rho, \sigma^T, \omega^T) \),

where \( O^T : \text{Op} \rightarrow \text{Taint} \times \text{OTaint} \rightarrow T^* \rightarrow T \) is the taint evaluator on primitive operators.

get handles accessing a key in a hash map as:

\[
O(\text{get})(\sigma, \omega)(x, s) = \omega(x)(s)
\]

\[
O^T(\text{get})(\sigma^T, \omega^T)(x, s) = \omega^T(x)(s).
\]

The rule for hash map update is also straightforward:
The function \( \hat{I} : \text{Prog} \rightarrow \hat{\Sigma} \) provides injection into the abstract state-space:

\[ \hat{I}(pr) = (pr, [], \bot, \bot, \bot, \bot, \text{halt}, \hat{t}_0), \]

where \( \hat{t}_0 \) is a designated initial abstract time.

The continuation-application helper \( \text{apply} : \text{Kont} \times D \times T \times \text{Store} \times \hat{\text{OStore}} \times \hat{\text{Taint}} \times \hat{\text{OTaint}} \times \text{Time} \rightarrow \Sigma \) completes the transition by dispatching on the type of continuation.

For let-based continuations:

\[
\text{apply} (\text{letk}(v, e, \rho, \kappa), d, d', \sigma, \omega, \sigma^T, \omega^T, t)
\]

\[
= (v', \rho', \sigma', \omega', \sigma'^T, \omega'^T, \kappa), \quad \text{where}
\]

\[
\alpha = \text{bindaddr}(v, t)
\]

\[
\rho' = \rho[v \mapsto a]
\]

\[
\sigma' = \sigma[a \mapsto d]
\]

\[
\sigma'^T = \sigma^T[a \mapsto d^T].
\]

and, for introspective iteration:

\[
\text{apply} (\text{fork}(v, v', e, \rho, [s_1 \mapsto d_1, \ldots]),
\]

\[
[k_1 \mapsto d'_1, \ldots, k_n \mapsto d_n], \sigma, \omega, \sigma^T, \omega^T, t)
\]

\[
= (v, \rho', \sigma', \omega, \sigma'^T, \omega'^T, \kappa), \quad \text{where}
\]

\[
\alpha = \text{bindaddr}(v, t)
\]

\[
\rho' = \rho[v \mapsto a, v' \mapsto a']
\]

\[
\sigma' = \sigma[a \mapsto s_1, a' \mapsto d_1]
\]

\[
\sigma'^T = \sigma^T[a \mapsto s_1^T, a' \mapsto d_1^T]
\]

\[
k' = \text{fork}(v, v', e, \rho, [s_2 \mapsto d_2, \ldots, [s_k \mapsto d_k], \ldots, k_n \mapsto d_n], \ldots, k_n \mapsto d_n, \ldots, \kappa).
\]

At this point, the definition of the remaining rules is straightforward, and follow the style of work done by Van Horn and Mitg (31).

4.2 Abstract semantics of HFTA

We now apply structural abstraction in full to the concrete state-space of the previous section. This structural abstraction yields a framework on top of which we can layer shape analysis for hash maps for further precision.

Applying a structural abstraction to the concrete state-space yields the abstract state-space in Figure 5.

The function \( \hat{I} : \text{Prog} \rightarrow \hat{\Sigma} \) provides injection into the abstract state-space:

\[ \hat{I}(pr) = (pr, [], \bot, \bot, \bot, \bot, \text{halt}, \hat{t}_0), \]

where \( \hat{t}_0 \) is a designated initial abstract time.

\[ \xi \in \Sigma = \text{Exp} \times \hat{\text{Env}} \times \hat{\text{Store}} \times \hat{\text{OStore}} \times \hat{\text{Taint}} \times \hat{\text{OTaint}} \times \text{Kont} \times \text{Time} \]

\[ \hat{\rho} \in \hat{\text{Env}} = \text{Var} \rightarrow \hat{\text{Addr}} \]

\[ \hat{\sigma} \in \hat{\text{Store}} = \text{Addr} \rightarrow \hat{\text{D}} \]

\[ \hat{\omega} \in \hat{\text{OStore}} = \hat{\text{OLoc}} \rightarrow \hat{\text{Obj}} \]

\[ \hat{\sigma}^T \in \hat{\text{Taint}} = \text{Addr} \rightarrow \hat{\text{T}} \]

\[ \hat{\omega}^T \in \hat{\text{OTaint}} = \hat{\text{OLoc}} \rightarrow \hat{\text{Obj}} \]

\[ d \in \hat{\text{D}} = \{\text{tainted, untainted}\}
\]

\[ \hat{\rho} \in \hat{\text{Env}} = \{\text{Var} \rightarrow \hat{\text{Addr}}\} \]

\[ \hat{\sigma} \in \hat{\text{Store}} = \{\text{Addr} \rightarrow \hat{\text{D}}\} \]

\[ \hat{\omega} \in \hat{\text{OStore}} = \{\text{OLoc} \rightarrow \hat{\text{Obj}}\} \]

\[ \hat{\omega}^T \in \hat{\text{OTaint}} = \{\text{Obj} \rightarrow \hat{\text{T}}\} \]

\[ \hat{\kappa} \in \hat{\text{Kont}} := \text{letk}(v, e, \hat{\rho}, \hat{\kappa}) \]

\[ \mid \text{fork}(v, v', e, \rho, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{\kappa}) \]

\[ \mid \text{halt} \]

\[ \hat{a} \in \hat{\text{Addr}} := \text{bindaddr}(v, i) \]

\[ \hat{t} \in \hat{\text{OLoc}} \] is a finite set of abstract object locations

\[ \hat{t} \in \hat{\text{Time}} \] is a finite set of abstract times

**Figure 5.** A parameterized abstract state-space.

4.2.1 Partial order of the abstract state-space

The partial order on the abstract state-space defined in Figure 6 lifts naturally element-wise, point-wise, member-wise and component-wise up from the leaves of the state-space: abstract locations, abstract denotable values, abstract integers and abstract times. (Abstract locations and abstract times must have a flat order.)

4.2.2 An explicit, structural abstraction

We can also define an explicit, structural abstraction from the concrete state-space into the abstract state-space (Figure 7).

The leaves of the state-spaces allow analysis-designers to tune precision (and speed) by varying the structure of their Galois connections.

Fixing a Galois connection over abstract denotable values, specifically, for abstract string, determines how the analysis reasons about dynamic string manipulation:

\[ (\mathcal{P}(\text{String}), \subseteq) \leq_{\alpha} (\tilde{\mathcal{P}}(\text{String}), \subseteq). \]

The Galois connection over integers determines reasoning over arithmetic:

\[ (\mathcal{P}(\mathbb{Z}), \subseteq) \leq_{\alpha} (\mathbb{Z}, \subseteq). \]

The Galois connection over times determines the context-sensitivity:

\[ (\mathcal{P}(\text{Time}), \subseteq) \leq_{\alpha} (\tilde{\mathcal{P}}(\text{Time}), \subseteq). \]

For a \( k \)-CFA-like notion of context-sensitivity, concrete times would become sequences of expressions, and abstract times would be \( k \)-tuples of expressions:

\[ \text{Time} = \text{Exp}^k \]

\[ \tilde{\text{Time}} = \text{Exp}^k. \]
(e, ρ, σ, ω, σ^T, ω^T, κ, i) ⊆ (e', ρ', σ', ω', σ'^T, ω'^T, κ', i')
iff e = e' and ρ ⊑ ρ' and σ ⊑ σ' and ω ⊑ ω'
and σ^T ⊑ σ'^T and ω^T ⊑ ω'^T and κ ⊑ κ' and i ⊑ i'

ρ ⊑ ρ' iff ρ(v) ⊑ ρ'(v) for all v ∈ dom(ρ)
σ ⊑ σ' iff σ(a) ⊑ σ'(a) for all a ∈ dom(σ)
ω ⊑ ω' iff ω(ℓ) ⊑ ω'(ℓ) for all ℓ ∈ dom(ω)
σ^T ⊑ σ'^T iff σ^T(ã) ⊑ σ'^T(ã) for all ã ∈ dom(σ^T)
ω^T ⊑ ω'^T iff ω^T(ℓ) ⊑ ω'^T(ℓ) for all ℓ ∈ dom(ω^T)

letk(v, e, ρ, κ) ⊆ letk(v', e', ρ', κ') iff v = v' and e = e'
and ρ ⊑ ρ' and κ ⊑ κ'.

fork(v1, v2, e, ρ, σ, σ^T, κ) ⊆ fork(v1', v2', e', ρ', σ'^T, κ')
iff v1 = v1' and v2 = v2' and e = e'
and ρ ⊑ ρ' and σ ⊑ σ' and σ^T ⊑ σ'^T and κ ⊑ κ'.

halt ⊑ hal

σ ⊑ σ' iff σ(ã) ⊑ σ'(ã) for all ã ∈ dom(σ)
σ^T ⊑ σ'^T iff σ^T(ã) ⊑ σ'^T(ã) for all ã ∈ dom(σ^T)

d ⊑ d' iff for all ù ∈ d there exists ù' ∈ d' such that ù ⊑ ù'

λ(ν, ρ) ⊑ λ(ν', ρ') iff lam = lam' and ρ ⊑ ρ'.

bindaddr(v, i) ⊆ bindaddr(v', i') iff v = v' and i = i'.

Figure 6. A partial order on abstract states.

so that abstraction will select the first k expressions:
α\{(e_1, \ldots, e_k)\} = (e_1, \ldots, e_k),

while concretization will append every possible tail:

γ(T) = \text{Exp}^*
γ(\ell) = \{\ell\} \times \text{Exp}^*
γ(\bot) = \emptyset.

The Galois connection over object locations determines object-polyvariance:

\[ (P(\text{OLoc}), \subseteq) \overset{\gamma}{\rightarrow} (\text{OLoc}^\top, \subseteq). \]

For a monovariant (OCFA-like) treatment of objects, the abstract location allocated for a new object would be the expression from where it came. To make this correct for the concrete semantics, concrete object locations would be expressions paired with the

\[ α(\{(e, t)\}) = e \]
\[ γ(T) = \text{OLoc} \]
\[ γ(\ell) = \{\text{expr}\} \times \text{Time} \]
\[ γ(\bot) = \emptyset. \]

4.2.3 Abstract transition rules

Our next step is to synthesize the abstract transition relation, \( (\rightsquigarrow) \subseteq \bar{Σ} \times \bar{Σ} \).

Most of the rules are straightforwardly adapted from the literature on flow analysis [31], so we focus on the ones that are novel for this work—the rules dealing with hash maps with taint analysis.

For allocating a hash map: Fig 8, where the expression \( \ell + 1 \) expands into the recording of execution context appropriate for the desired context-sensitivity, and the abstract object allocator

\[ \text{alloc} : \bar{Σ} \to \text{OLoc} \]

returns the abstract location for this object.

To match up with the earlier OCFA-like object-polyvariance example, the correct allocator in this case would return the current ex-
The abstract argument evaluator $\hat{\text{AExp}} \times \hat{\text{Env}} \times \hat{\text{Store}} \times \hat{\text{OSTore}} \rightarrow \hat{\mathcal{D}}$ mimics its concrete counterpart:

$\hat{\text{A}}(e, \hat{\rho}, \hat{\sigma}, \hat{\omega}) = \alpha(e)$

$\hat{\text{A}}(v, \hat{\rho}, \hat{\sigma}, \hat{\omega}) = \delta(\hat{\rho}(v))$

$\hat{\text{A}}(\text{lam}(\hat{\lambda}, \hat{\rho}, \hat{\sigma}, \hat{\omega}) = \{(\hat{\lambda}, \hat{\rho})\}$

$\hat{\text{A}}((\text{op } x_1 \ldots x_n), \hat{\rho}, \hat{\sigma}, \hat{\omega}) = \hat{\text{O}}(\text{op})(\hat{\sigma}, \hat{\omega})(d_1, \ldots, d_n)$

where $d_i = \hat{\text{A}}(x_i, \hat{\rho}, \hat{\sigma}, \hat{\omega})$.

The abstract taint argument evaluator $\hat{\text{AExp}} \times \hat{\text{Env}} \times \hat{\text{Taint}} \times \hat{\text{OExp}} \rightarrow \hat{\mathcal{T}}$ is like $\hat{\text{A}}$, but evaluates taintedness:

$\hat{\text{A}}^T(e, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\mathcal{T}}) =$ untainted

$\hat{\text{A}}^T(v, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\mathcal{T}}) = \hat{\sigma}^T(\hat{\rho}(v))$

$\hat{\text{A}}^T(\text{lam}(\hat{\lambda}, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\mathcal{T}}) =$ untainted

$\hat{\text{A}}^T((\text{op } x_1 \ldots x_n), \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\mathcal{T}}) = \hat{\text{O}}^T(\text{op})(\hat{\sigma}, \hat{\omega})(d_1, \ldots, d_n)$

where $d_i = \hat{\text{A}}(x_i, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\mathcal{T}})$.

The abstract continuation-application auxiliary function $\hat{\text{apply}} : \hat{\text{Kont}} \times \hat{\mathcal{D}} \times \hat{\mathcal{T}} \times \hat{\text{Store}} \times \hat{\text{OSTore}} \times \hat{\text{Taint}} \times \hat{\text{OExp}} \times \hat{\text{Time}} \rightarrow \hat{\mathcal{P}}(\hat{\Sigma})$ considers the application of a continuation. In the case of completing of let-bound values and taint property:

$\hat{\text{apply}}(\text{letk}(v, e, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{\mathcal{T}}))
= (e, \hat{\rho}', \hat{\sigma}', \hat{\omega}', \hat{\sigma}^T, \hat{\omega}^T, \hat{\mathcal{T}}, \hat{\mathcal{K}})$

where $\hat{\alpha} = \text{bindaddr}(v, \hat{\sigma})$

$\hat{\rho}' = \hat{\rho}(v \mapsto \hat{\alpha})$

$\hat{\sigma}' = \hat{\sigma}[\hat{\alpha} \mapsto \hat{\sigma}' \hat{\mathcal{T}}]$

$\hat{\omega}' = \hat{\omega} \cup \{\hat{\alpha} \mapsto \hat{\sigma}'\}$

and in the case of for introspective iteration:

$\hat{\text{apply}}(\text{fork}(v, v', e, \hat{\rho}, [\hat{\sigma}_1 \mapsto \hat{\sigma}_2, \ldots])$

$\begin{align*}
\{e, \hat{\rho}', \hat{\sigma}', \hat{\omega}', \hat{\sigma}^T, \hat{\omega}^T, \hat{\mathcal{T}}, \hat{\mathcal{K}}'\}
\end{align*}$

$\hat{\text{K}}' = \{\text{fork}(v, v', e, \hat{\rho}, [\hat{\sigma}_1 \mapsto \hat{\sigma}_2, \ldots], [\hat{\sigma}_1' \mapsto \hat{\sigma}_2', \ldots], \hat{\mathcal{K}}')\}$.

The jarring dissonance of this rule for the function $\hat{\text{apply}}$ with respect to its counterpart in the concrete $\text{apply}$ is notable. According to this rule, after every iteration of the abstract loop, it will simulate (1) leaving the loop, (2) advancing in the loop and (3) repeating what appears to be the same element.

But, why?

This extreme conservatism is forced upon the analysis by a lack of shape-analytic information regarding the hash map.

A simple example highlights why this conservatism is necessary:

$(\text{foreach } k \text{ in } o$ $ \text{(set! i } (\star i 1)) )$

In this example, the loop counts the number of properties in the hash map.

With a sufficiently precise abstraction on integers, the abstract interpretation will contain a path that computes the precise count.

But, if the key to an abstract hash map is imprecise, then the analysis may not know how many keys it represents, and thus, how many times to execute the abstract loop. To remain sound, it must always jump back on itself. This important subtlety, and a desire to improve the precision of analysis of introspective iteration over hash maps, is what motivates the generalization of shape-analytic techniques to hash maps in the next sections.

4.2.4 Proof of soundness

The formulation of the soundness theorem is standard; the abstract transition must simulate the concrete transition:

Theorem 1. If:

$\alpha(\zeta) \subseteq \xi \text{ and } \zeta \Rightarrow \zeta'$.
then there must exist an abstract state $\xi'$ such that: $\alpha(\xi') \sqsubseteq \xi'$ and $\xi \leadsto \xi'$.

Proof. Assume $\alpha(\xi) \sqsubseteq \xi$ and $\xi \Rightarrow \xi'$. The high-level structure of the proof splits on cases with respect to the expression within $\xi$. The structure of each case mimics the straightforward style found in [16].

4.2.5 Computation of the analysis

As noted earlier, to compute the analysis as a classical small-step flow analysis, we must first thread continuations through the store in order to bound the height of the lattice for the abstract state-space. To do this in the style of Van Horn and Migh [31], reformulate continuations to contain pointers to continuations rather than continuations themselves:

$$\hat{k} \in \text{Kont} ::= \text{letk}(v, e, \hat{\rho}, \hat{a})$$

$$| \text{fork}(v_{\text{key}}, v_{\text{value}}, e, \hat{\rho}, \hat{a} \hat{T}, \hat{a})$$

$$| \text{halt}$$

At this point, analysis proceeds by injecting a program $pr$ into an initial abstract state:

$$\zeta_0 = I(pr),$$

and then finding the states reachable from this state:

$$\{ \zeta : \zeta_0 \Rightarrow^{*} \zeta \}.$$  

Another potential source of unboundedness in the abstract state-space is the abstract domain for strings. Many abstract domains for strings are infinite in height, which means widening is required to achieve termination. Each abstract domain for strings requires its own widening criteria and techniques.

4.3 Cardinality analysis of abstract hash maps

For ordinary objects and pointers, a cardinality analysis [4, 18, 19] enables strong updates (and deletions) during the analysis. With the refactoring into a Curried object store, we can generalize a cardinality analysis to keys in a hash map. The key step is to conduct a second, parallel abstract interpretation, $\alpha^\omega : \Sigma \rightarrow O\text{Store}^\omega$, that measures the cardinality of abstract locations:

$$\omega^\omega \in O\text{Store}^\omega = O\text{Loc} \rightarrow \{ 0, 1, \infty \}$$

$$\alpha^\omega(e, \rho, \sigma, \sigma^T, \omega^T, \kappa, t) = \lambda \hat{\ell}.\text{size}(\gamma(\hat{\ell}) \cap \text{dom}(\omega)).$$

where:

$$\text{size} \{ \} = 0$$

$$\text{size} \{ x \} = 1$$

$$\text{size} \{ x_1, x_2, \ldots, x_n \} = \infty.$$  

We then formulate the reduced product transition of the abstractions $\alpha \times \alpha^\omega$, which effectively adds a $\omega^\omega$ component to every state. Having this cardinality information available makes it possible to perform strong update.

The reduced abstract transition rule for hash map update demonstrates strong update on both objects and on individual keys:

$$\Rightarrow^\omega$$

$$\Rightarrow^\omega$$

$$\Rightarrow^\omega$$

$$\Rightarrow^\omega$$

$$\Rightarrow^\omega$$

$$\Rightarrow^\omega$$

5. Implementation and evaluation

We have implemented the analytical framework HFTA for $\lambda_H$ using Racket. To demonstrate the feasibility and effectiveness of HFTA, we did our experimentation for large subset of Python 3.0 and are able to precisely detect security vulnerabilities, like XSS for web applications written in Python.

The architecture is illustrated in Figure 5. Specifically, after tokenizing source programs by PyParser then the generation of the abstract syntax tree by PyParser, we semantic-equivalently transformed Python code into Lisp/Scheme-like language by PyTranslator, which compiles classes into closures and hash maps, instances into hash maps, fixes scoping rules like global, nonlocal, and desugars return using call/cc.
We experimented further with the context $k = 0$ and $k = 1$ with cardinality analysis presented in 4.3, as shown in Table 2. There is a small percentage reduction of analysis time when $k = 0$. (This agrees with the experimental results on context-sensitivity in [20].) To demonstrate the effectiveness of the cardinality analysis on the Curried object store, Table 3 shows the precision of the analysis both with and without the cardinality analysis enabled.  

5.2 Case Study

In this section, we take a closer look at the search engine benchmark to demonstrate the feasibility and effectiveness of tracking taint data flows to find vulnerabilities using our analytic framework.

The analyzer is implemented as k-CFA CESK-style abstract interpreter, which is almost translating the specification of small-step abstract semantics presented in section 4.2 (especially 4.2.3) of HFTA and the work [31] into actual code.

5.1 Preliminary Results

Series of experiments have been conducted on simple web applications written in Python. We did all these benchmarks on a machine with an Intel Core i7 930 2.8GHz CPU, 8GB DDR3 RAM, and a 80GB Intel X25-M SSD. Table 1 presents the overall results.

The first four are simple web applications but in object-oriented style with hash-table of first-class procedures: factorial (wfib), Fibonacci (wfib), color chooser (cc) and a converter (conv). The analyzer is implemented using the algorithm [14].

Another subprocess in the translation process is desugaring, which further desugars constructs like break, continue, all the looping variants (while, for, for/else), branching variants using call/cc. We also desugars container comprehension operations since they are used a lot in practice. How to use first class continuation do desugaring is done basically in the way as [15]. The A-Normalizer is implemented using the algorithm [20].

The XSS attacks reported are previously unknown vulnerabilities. They resides in sensitive sinks, like printin or write. The effectiveness of taint analysis in our framework is specifically discussed in Section 5.2.

Fig 10 is the simplified version of the search engine program for expository purpose. It reads in the search terms extracted from cgi form, which is sanitized (or not) then sent to the search algorithm to compute the result. If found, then the program will render search results. Otherwise, echo back the not found page.

To be able to precisely and effectively analyze web program like this, our analysis not only explicitly propagated taint information, such as simple variable references/assignments, field/keys access

The metric\(^{[i]}\) of\(\text{objects}\) is used as a crude metric for field sensitivity of objects (including hash maps, container like list, set and tuple, objects). It is a ratio between \(\[i]\) and \(\text{objects}\), where \(i = 1, 2, \ldots\) is the number of abstract location (objects) with a cardinality of \(i\), and \(\text{objects}\) is total number of objects in a program. The larger the ratio of a smaller cardinality of objects, the more precise and field sensitive the analysis. In particular, we are most interested in the case where the cardinality \(i = 1\), where is proofed sound to do strong updates for shape analysis. In Table 1, the row metric\(^{[i]}\) of\(\text{objects}\) indicates reasonable precision even for five preliminary analysis.

Table 1. Preliminary Results of HFTA with $k = 1$ and cardinality analysis
With Cardi No-Cardi

<table>
<thead>
<tr>
<th>Prog</th>
<th>With-Cardi</th>
<th>No-Cardi</th>
</tr>
</thead>
<tbody>
<tr>
<td>wtb</td>
<td>0.529</td>
<td>0.661</td>
</tr>
<tr>
<td>wfact</td>
<td>0.268</td>
<td>0.322</td>
</tr>
<tr>
<td>cc</td>
<td>0.244</td>
<td>0.293</td>
</tr>
<tr>
<td>conv</td>
<td>0.072</td>
<td>0.078</td>
</tr>
<tr>
<td>srch</td>
<td>12</td>
<td>14.2</td>
</tr>
</tbody>
</table>

Table 2. CPU time (sec) of HTFA for context-sensitivity (with cardinality analysis)

<table>
<thead>
<tr>
<th>Prog</th>
<th>With-Cardi</th>
<th>No-Cardi</th>
</tr>
</thead>
<tbody>
<tr>
<td>weblib</td>
<td>56%</td>
<td>44%</td>
</tr>
<tr>
<td>webfact</td>
<td>69%</td>
<td>53%</td>
</tr>
<tr>
<td>cc</td>
<td>68%</td>
<td>55%</td>
</tr>
<tr>
<td>conv</td>
<td>57%</td>
<td>57%</td>
</tr>
<tr>
<td>srch</td>
<td>55%</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 3. Field-sensitivity (with $k = 1$)

and updates in objects/hash maps, under arbitrary levels of nesting, but also achieved implicit taint propagation due to control flows, such as conditional statements, function/method calls and returns, and etc, which are formally specified and illustrated in section 4.2.3.

Specifically in the code sample at line 4-10, the tainted input of s1 at line 41 and s2 at line 45 will flow into the constructor of Search at line 4 and then Request’s data field at line 15, resp’s result at line 8, as well as resp’s body field at line 30 and line 34. That is, unsanitized input flows back to the client at line 11, creating the possibility of a an XSS attack.

To reduce false alarms, we manually wrote the annotation file that will be pre-loaded and processed by our analyzer, to “detaint” tainted values, that is, return values from the escape in cgi module, Python’s parse_request in HTTP server library will be marked untaint. So at line 46, if s2 were meant to invoke sanitizing process at line 46– the handler that is pre-installed in Request’s hash map at line 16, the call at line 47 will not trigger an XSS alarm, since the data is sanitized, despite of the XSS sensitive sink that renders content to screen. While line 42 will trigger the alarm, since the content to be displayed is tainted without any sanitization.

5.3 Discussion

Currently, the analyzer for Python does not support eval, exec; these are not considered as good practice in most scripting languages and reported are seldom used in real applications [25, 28].

Our framework is designed to analyze dynamic languages, and in practice, if one wishes to analyze a specific scripting language with our framework, one needs to develop the front-end translation into lambda. For example, in our experimental prototype for Python, all of Python’s metaprogramming features (reflection) are compiled into hash maps in lambda.

6. Related work

The overall approach of this work was to augment a powerful analysis of higher-orderness with a rich analysis of dynamic hash maps. One could easily ask why not go the other direction—take a rich shape analysis of hash maps in a first-order language and augment it with higher-orderness? Or, in fact, to encode closures flatly as dynamic objects? Work by MIGHT, SMARAGDAKIS and VAN HORN [20] discusses the relationship between static analysis of higher-order functions and objects. It turns out that encoding closures (flatly) as objects sacrifices opportunities for precision, since is strictly contracts the abstract state-space. Analyzing closures directly allows for a nested abstraction that better separates individual bindings to variables.

In the following, we mainly compare HFTA with other work in the area of static approach for taint analysis. Related work for dynamic taint analysis can be found [3]. Existing approaches to static analysis of taints usually have one of two problems. The first problem is limited language feature supported. Pixy [11] is a static taint analysis for PHP that propagates taint information and implements finely tuned alias analysis; Xie and Aiken designed a more precise and scalable analysis for detecting SQL Injection attacks in PHP by using block- and function-summaries. Unfortunately, both of these can’t be used to track taint information in objects or hash maps. This is also the case for other work, such as [21, 34, 35], and one of the few static analysis for Python [28].

The second problem is the lack of soundness in existing approaches [30]. Realizing tracking data-flow through hash maps is difficult, [30] resolves the problem with the assumption of constant hash keys, an assumption quickly violated by many real programs. Moreover, the “nested taint depth” is tuned to be at most two levels, which, once again, exceeds the behaviors commonly experienced in modern web applications.

Our approach tackles both problems simultaneously. It tackles robustness with respect to complex language features by constructing a systematic abstraction of an abstract machine for a core calculus of those language features. The same systematic approach—the systematic conversion of an abstract machine into an abstract interpreter—simultaneously guarantees soundness, thereby tackling the second problem.

7. Conclusion and Future work

In this work, we provide a systematic approach to the static taint analysis for web programs written in scripting languages—even in the presence of dynamic hash maps, higher-order functions, and first-class control. The ability to analyze these features in tandem is a critical step towards practical static analyzers for modern web applications, not just for static taint analysis, but for deep static analysis of these languages in general.

We have implemented our framework for Python, conducted preliminary experiments with small web applications. By achieving a low rate of manually resolvable false alarms, we demonstrated the promise, feasibility and effectiveness of our approach.

In future, we are going to carry out more precise and specific analysis on top of HFTA. The first one is a must-contain key-shape analysis of hash maps, similarly as cardinality shape analysis of abstract hash maps but with more precise information to compute dynamic keys that must be within a map, in addition to knowing which may be within the map. This can further improve precision of field-sensitivity for both hash maps and dynamic objects. The other direction we will pursue is the static string analysis based on our framework. Though the analysis challenge exposed by the micro-benchmarks are tackled by our current analytic framework, we are considering to compile larger set of Python and libraries to practice the analyzer on larger web applications in Python. We are also integrating Abstract Garbage Collection technique [18, 19]
into our analytic framework for scalability improvement, which is reported to significantly reduce analysis complexity.

Acknowledgments

We thank our anonymous reviewers for their detailed comments on the submitted paper. We especially thank Avik Chaudhuri for his guidance in preparing the final version of the paper. This material is based on research sponsored by DARPA under the programs Automated Program Analysis for Cybersecurity (FA8750-12-2-0106) and Clean-Slate Resilient Adaptive Hosts (CRASH). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Disclaimer

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References